Finding the shortest route on graph with Julia JuMP

Przemysław Szufel
https://szufel.pl/

Use case scenario

The Subway restaurant chain in Las Vegas has a total of 118 restaurants in different parts of the city.

Company's manager plans to visit all restaurants during a single day.

What is the optimal order that restaurants should be visited?

Traveling salesman problem

- Variables:
  - c_{ft} – cost of travel from “f” to “t”
  - x_{ft} – binary variable indicating 1 when agent travels from “f” to “t”

$$\text{Min} \sum_{f=1}^{N} \sum_{t=1}^{N} c_{ft} x_{ft}$$

The mathematical problem framing based on: Julia Programming for Operations Research A Primer on Computing, Changhyun Kwon, 2018, see: http://opensourc.es/blog/mip-tsp

TSP

Variables:

- c_{ft} – cost of travel from "f" to "t"
- x_{ft} – binary variable indicating 1 when agent travels from "f" to "t"

$$\mathbf{Min} \sum_{f=1}^{N} \sum_{t=1}^{N} c_{ft} x_{ft}$$

Each city visited once

$$\sum_{t=1}^{N} x_{ft} = 1 \quad \forall f \in \{1, \ldots, N\}$$

$$\sum_{f=1}^{N} x_{ft} = 1 \quad \forall t \in \{1, \ldots, N\}$$

City cannot visit itself

$$x_{ff} = 0 \quad \forall f \in \{1, \ldots, N\}$$

Avoid two-city cycles

$$x_{ft} + x_{tf} <= 1 \quad \forall f, t \in \{1, \ldots, N\}$$

Other cycles: /dynamically add a constraint whenever a cycle occurs/

For more details see: http://opensourc.es/blog/mip-tsp

JuMP implementation

```
m = Model(with_optimizer(GLPK.Optimizer))
@variable(m, x[f=1:N, t=1:N], Bin)
@objective(m, Min, sum( x[i, j]*distance_mx[i,j] for i=1:N,j=1:N))
@constraint(m, notself[i=1:N], x[i, i] == 0)
@constraint(m, oneout[i=1:N], sum(x[i, 1:N]) == 1)
@constraint(m, onein[j=1:N], sum(x[1:N, j]) == 1)
for f=1:N, t=1:N
    @constraint(m, x[f, t]+x[t, f] <= 1)
end
```

Getting a cycle

```
function getcycle(m, N)
    x_val = value.(x)
    cycle_idx = Vector{Int}()
    push!(cycle_idx, 1)
    while true
        v, idx = findmax(x_val[cycle_idx[end], 1:N])
        if idx == cycle_idx[1]
            break
        else
            push!(cycle_idx, idx)
        end
    end
    cycle_idx
end
```

Adding a constraint…

```
function solved(m, cycle_idx, N)
    println("cycle_idx: ", cycle_idx)
    println("Length: ", length(cycle_idx))
    if length(cycle_idx) < N
        cc = @constraint(m, sum(x[cycle_idx,cycle_idx]) 
<= length(cycle_idx)-1)
        println("added a constraint")
        return false
    end
    return true
end
```

Iterating over the model

```
while true
    optimize!(m)
    println(termination_status(m))
    cycle_idx = getcycle(m, N)
    if solved(m, cycle_idx,N)
        break;
    end
end
```

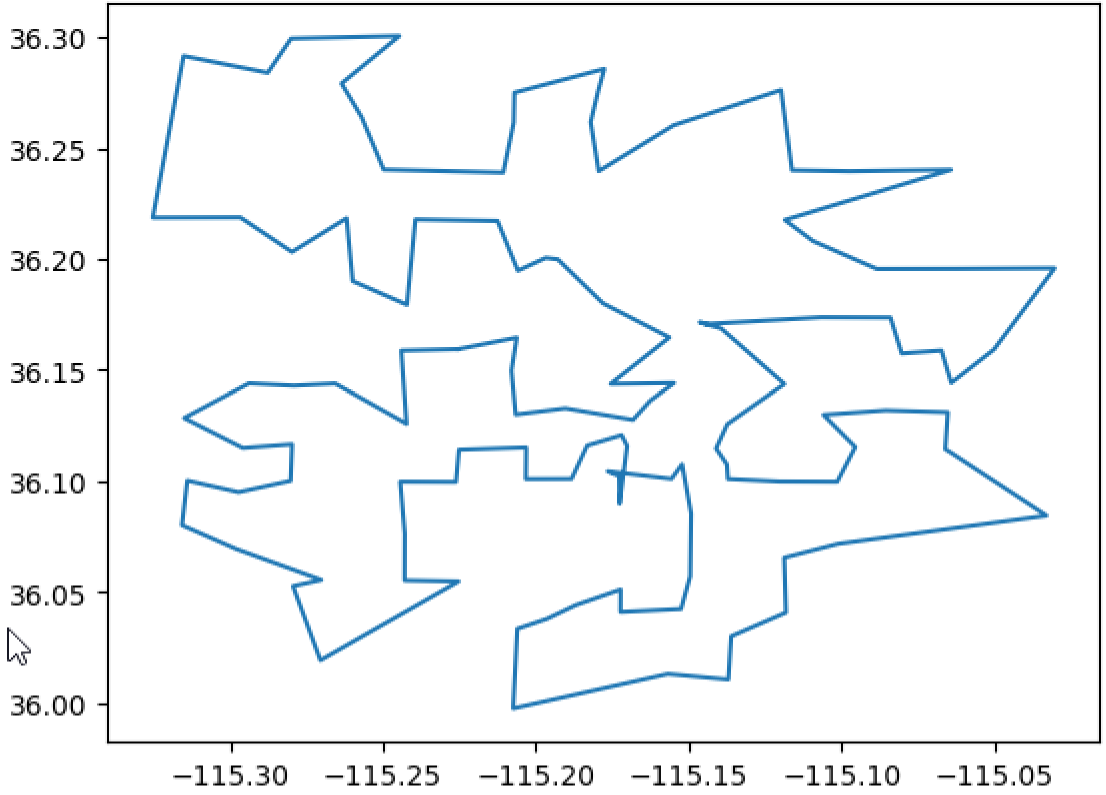
JuMP - available solver packages

Solver	Julia Package	License	LP	SOCP	MILP	NLP	MINLP	SDP
Artelys Knitro	KNITRO.jl	Comm.				X	X	
BARON	BARON.jl	Comm.				X	X	
Bonmin	AmplNLWriter.jl CoinOptServices.jl	EPL	X		X	X	X	
Cbc	Cbc.jl	EPL			X			
Clp	Clp.jl	EPL	X					
Couenne	AmplNLWriter.jl CoinOptServices.jl	EPL	X		X	X	X	
CPLEX	CPLEX.jl	Comm.	X	X	X			
ECOS	ECOS.jl	GPL	X	X				
FICO Xpress	Xpress.jl	Comm.	X	X	X			
GLPK	GLPKMathProgInterface	GPL	X		X			
Gurobi	Gurobi.jl	Comm.	X	X	X			
Ipopt	Ipopt.jl	EPL	X			X		
MOSEK	Mosek.jl	Comm.	X	X	X	X		X
NLopt	NLopt.jl	LGPL				X		
SCS	SCS.jl	MIT	X	X				X

TravelingSalesmanHeuristics.jl

```
using TravelingSalesmanHeuristics

sol = TravelingSalesmanHeuristics.solve_tsp(
distance_mx,quality_factor =100)
```



More info:
http://evanfields.github.io/TravelingSalesmanHeuristics.jl/latest/heuristics.html

Background info - how JuMP works

Understand JuMP - metaprogramming

```
julia> code = Meta.parse("x=5")
:(x = 5)

julia> dump(code)
Expr
  head: Symbol =
  args: Array{Any}((2,))
    1: Symbol x
    2: Int64 5

julia> eval(code)
5

julia> x
5
```

Julia macros – hello world...

```
macro sayhello(name)
	return :( println("Hello, ", $name) )
end

julia> macroexpand(Main,:(@sayhello("aa")))
:((Main.println)("Hello, ", "aa"))

julia> @sayhello "world!"
Hello, world!
```

what is the @variable macro in JuMP

```
julia> @macroexpand @variable(m, x₁ >= 0)
quote
    (JuMP.validmodel)(m, :m)
    begin
        #1###361 = begin
                let
                    #1###361 = (JuMP.constructvariable!)(m, getfield(JuMP, Symbol("#_error#107")){Tuple{Symbol,Expr}}((:m, :(x₁ >= 0))), 0, Inf, :Default, (JuMP.string)(:x₁), NaN)
                    #1###361
                end
            end
        (JuMP.registervar)(m, :x₁, #1###361)
        x₁ = #1###361
    end
end
```

Why JuMP is fast - symbolic computing in Julia

Stop approximating derivatives!

Derivatives are required at the core of many numerical algorithms. Unfortunately, they are usually computed *inefficiently* and *approximately* by some variant of the finite difference approach

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, h \text{ small} .$$

This method is *inefficient* because it requires $\Omega(n)$ evaluations of $f : \mathbb{R}^n \to \mathbb{R}$ to compute the gradient $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \cdots, \frac{\partial f}{\partial x_n}(x) \right)$, for example. It is *approximate* because we have to choose some finite, small value of the step length h, balancing floating-point precision with mathematical approximation error.

What can we do instead?

One option is to explicitly write down a function which computes the exact derivatives by using the rules that we know from Calculus. However, this quickly becomes an error-prone and tedious exercise. **There is another way!** The field of automatic differentiation provides methods for automatically computing *exact* derivatives (up to floating-point error) given only the function f itself. Some methods use many fewer evaluations of f than would be required when using finite differences. In the best case, **the exact gradient of f can be evaluated for the cost of $O(1)$ evaluations of f itself**. The caveat is that f cannot be considered a black box; instead, we require either access to the source code of f or a way to plug in a special type of

`Calculus.jl` – symbolic computing

```
julia> using Calculus

julia> differentiate(:(sin(x)))
:(1 * cos(x))

julia> expr = differentiate(:(sin(x) + x*x+5x))
:(1 * cos(x) + (1x + x * 1) + (0x + 5 * 1))

julia> x = 0; eval(expr)
```