MIP - Travelling Salesman

Creation date: 2017-10-15

Tags: julia, mip, constraint-programming, linear-programming, optimization

Update: 15.10.2019 to Julia 1.2 and JuMP v0.20

Sometimes you're working on a project for too long and just need a funny little break. In the last article I mentioned numerical error rounding problems. I tried to fix them using various methods. Probably the best for it is the revised simplex method. There you're working with the initial tableau all the time which reduces the errors.

I tried to implement that and the functionality of adding constraints. I got stuck somewhere but I'll try to resolve it one day. Well I don't get paid for this so I do whatever I want. Today I want to show you what we can do with linear programming.

I should tell you a bit about Mixed Integer Programming (MIP) first. The solution you get after solving a linear program using the simplex method is real. In a lot of problems we want to have Integer solutions and a lot of times Binary values.

The solution we get using the simplex method can be a good approximation and the objective we get is definitely a lower bound/upper bound for our problem.

Let's have a look at a small problem

$$\begin{aligned} \max \quad & x + 2y & & \\ \text{subject to} \quad & x + y \leq 3.5 & & (1) \\ & y \leq 2 & & \\ & x, y \geq 0 & & \end{aligned}$$

Here the solution would be $y = 2, x = 1.5$ with an objective of 5.5.

If we want to have integer solutions we reformulate the problem as:

$$\begin{aligned} \max \quad & x + 2y & & \\ \text{subject to} \quad & x + y & \leq & 3.5 \\ & y & \leq & 2 \\ & x, y \in \mathbb{N} & & \end{aligned} \tag{2}$$

The solution to these kind of problems is normally to solve them without the extra $\in \mathbb{N}$ and try to fix that part later. Without considering the integer formulation the problem is called the linear relaxation. Relax first and then figure out the other part later on.

What do we have: $y = 2, x = 1.5$ okay alright. Doesn't look too bad. y is already integer. If we have a real value here 1.5 we can formulate two new problems. We can add a constraint: $x \geq 2$ and solve that and one problem where we add the constraint $x \leq 1$. Then x will be integer. That's correct it will be definitely integer but the other variables might be fractional even if they are integer before.

Let's solve $x \geq 2$. Well that reduces y to 1.5 to make the first constraint feasible again. The objective would be $2 + 2 * 1.5 = 5$ but we still don't have an Integer solution.

Solving the problem $x \leq 1$ we get $x = 1, y = 2$ with an objective of 5. This solution is integer and it can't be improved because the linear relaxation using $x \geq 2$ is also 5. Therefore we don't have to consider that problem anymore.

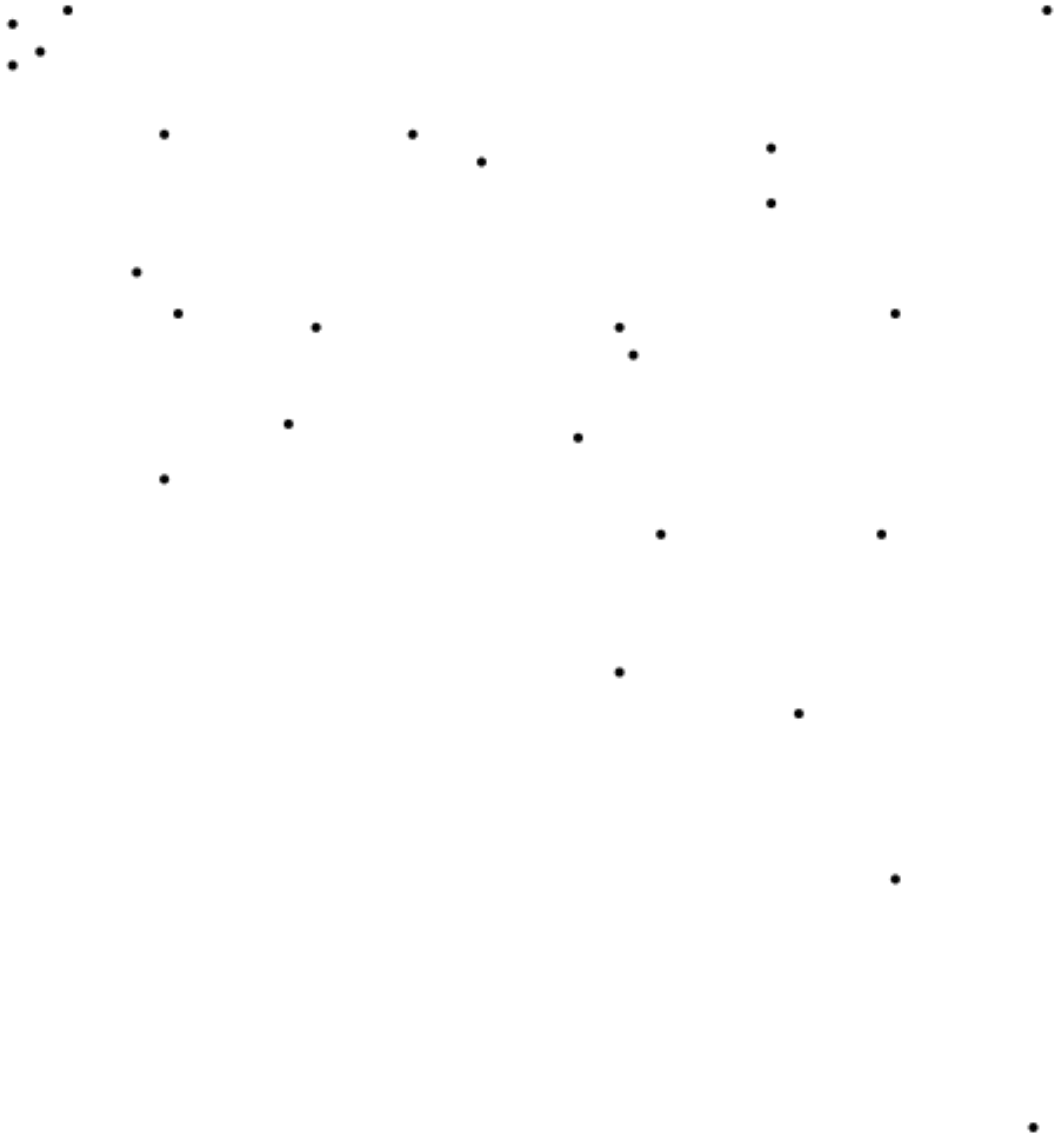
This technique is called Branch and Bound. Branch because we split the problem into several branches and bound because we can cut off branches if we know that they can't improve our solution.

Let's start with the real shit :D

Everyone who is reading about optimization stuff should know about the Travelling Salesman Problem (TSP)

The problem is the following: You're a salesman and you have to get to several cities to sell your product. You want to visit every city once. At the end of your trip you want to come back to your starting city. Therefore the path is a cycle. It costs time and money to travel so you want to use the shortest possible route.

We have the following map of cities.



The list of coordinates looks like this:

COPY

```
1 2
3 4
5 1
1 5
10 20
30 10
12 10
21 31
12 35
76 1
42 32
23 24
56 11
35 12
56 15
65 64
45 24
65 23
46 26
75 82
```

```
64 39
58 52
48 39
45 49
13 23
```

This time I'll use Julia for this task. It's quite an interesting programming language and is quite easy to read for optimization problems.

At first we have to include JuMP which is a modelling language for mathematical optimization in Julia. We need a solver which actually does the work. I use GLPK and also a library to calculate the distances between our points. It's called Distances

```
using JuMP, GLPK, Distances
```

COPY

```
file_name = "tsp_25_1"
f = open("./data/" * file_name);
lines = readlines(f)
N = length(lines)
println("N: ", N)
```

The file were the coordinates are stored is `tsp_25_1` because it has 25 cities. We get the lines of the files and store the number of cities in `N`.

Let's store the rest of the data in a vector.

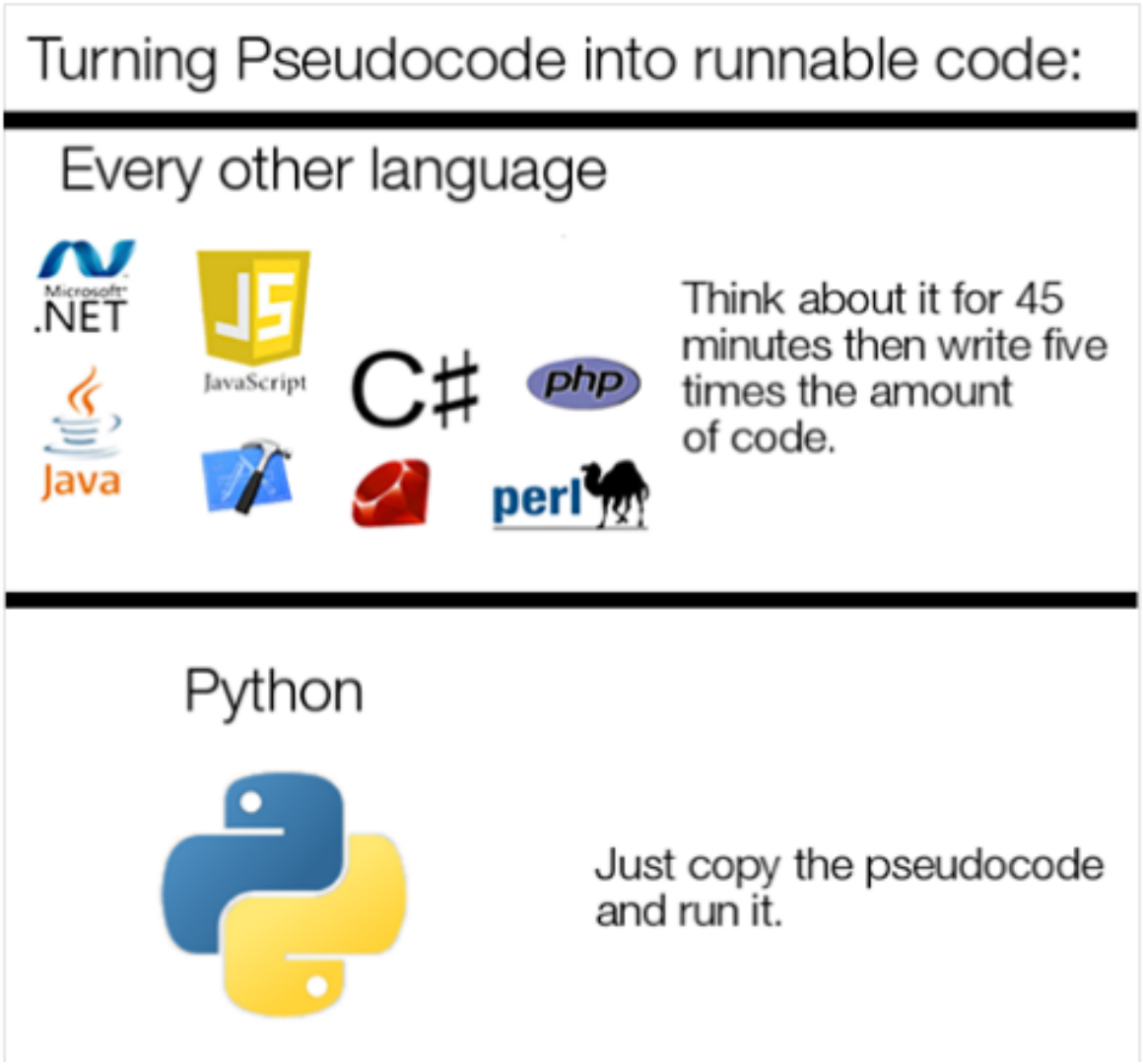
```
c_pos = [Vector{Float64}(undef, 2) for _ in 1:N]
```

COPY

```
for i = 1:N
    x_str, y_str = split(lines[i])
    c_pos[i] = [parse(Float64, x_str), parse(Float64, y_str)]
end
```

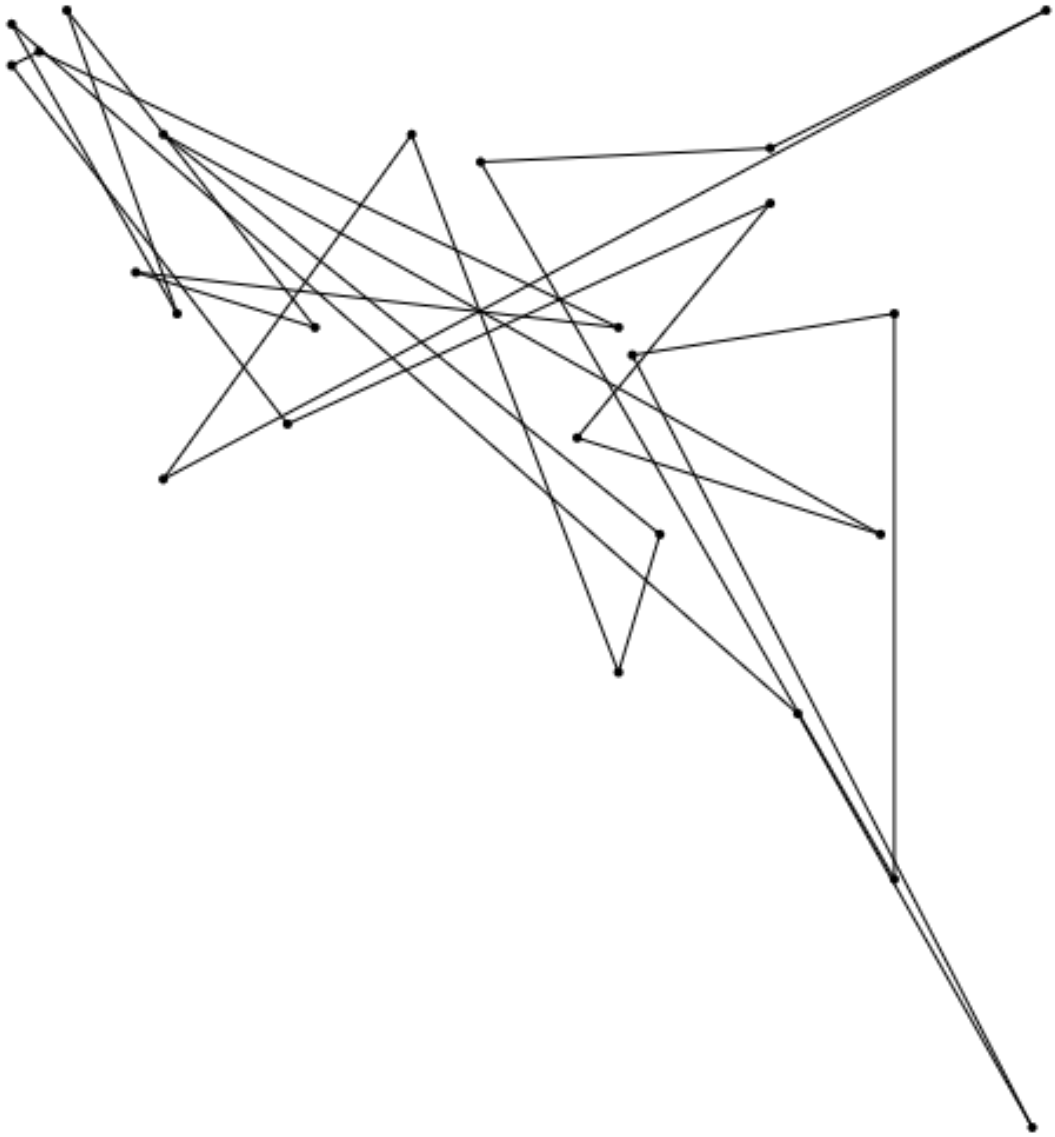
I think it's quite easy to understand that lines. Well I should mention that indices in Julia starts with 1 instead of 0.

If only python was 1 indexed to make this a simpler process 🤔



Source: Computer Science Memes for Travelling Salesman Teens on Facebook

Okay I think we should think about solving the TSP for a moment. Is it easily solveable? Well we can connect every city to every other city which gives us a lot of possibilites. Okay to get a valid solution we can connect the first point to one of 24 other cities the next one to the remaining 23 and so on. This gives us $24! = 620,448,401,733,239,439,360,000$ options. I don't know how you think about that but I think that number is huge. Most of these possibilities are pretty strange and it's obvious that they aren't optimal. Actually we can divide this by 2 because the route $A \to C \to B \to A$ is the same as $A \to B \to C \to A$. It's still huge.



Is it possible to solve it using a linear problem formulation? What variables do we have?

Let's make a variable for every possible connection of two cities. If we want to connect these cities the variable will be set to `1` and `0` otherwise. We could say that if A is connected to B then B is connected to A which reduces the number of options as mentioned above. For simplicity reasons I stick to the easier version here. Feel free to change the code to improve the performance ;)

That would be an Array with 25 rows and 25 columns which are 625 variables. That's not too bad.

The objective function is quite easy then: Minimize the distance of the connections we selected.

$$\text{Min} \sum_{f=1}^{N} \sum_{t=1}^{N} c_{ft} x_{ft} \tag{3}$$

The cost to get from one city to another is c_{ft} where `f` is the index of the city we are coming from and `t` the index of the city we are getting to. x_{ft} is the binary variable mentioned before.

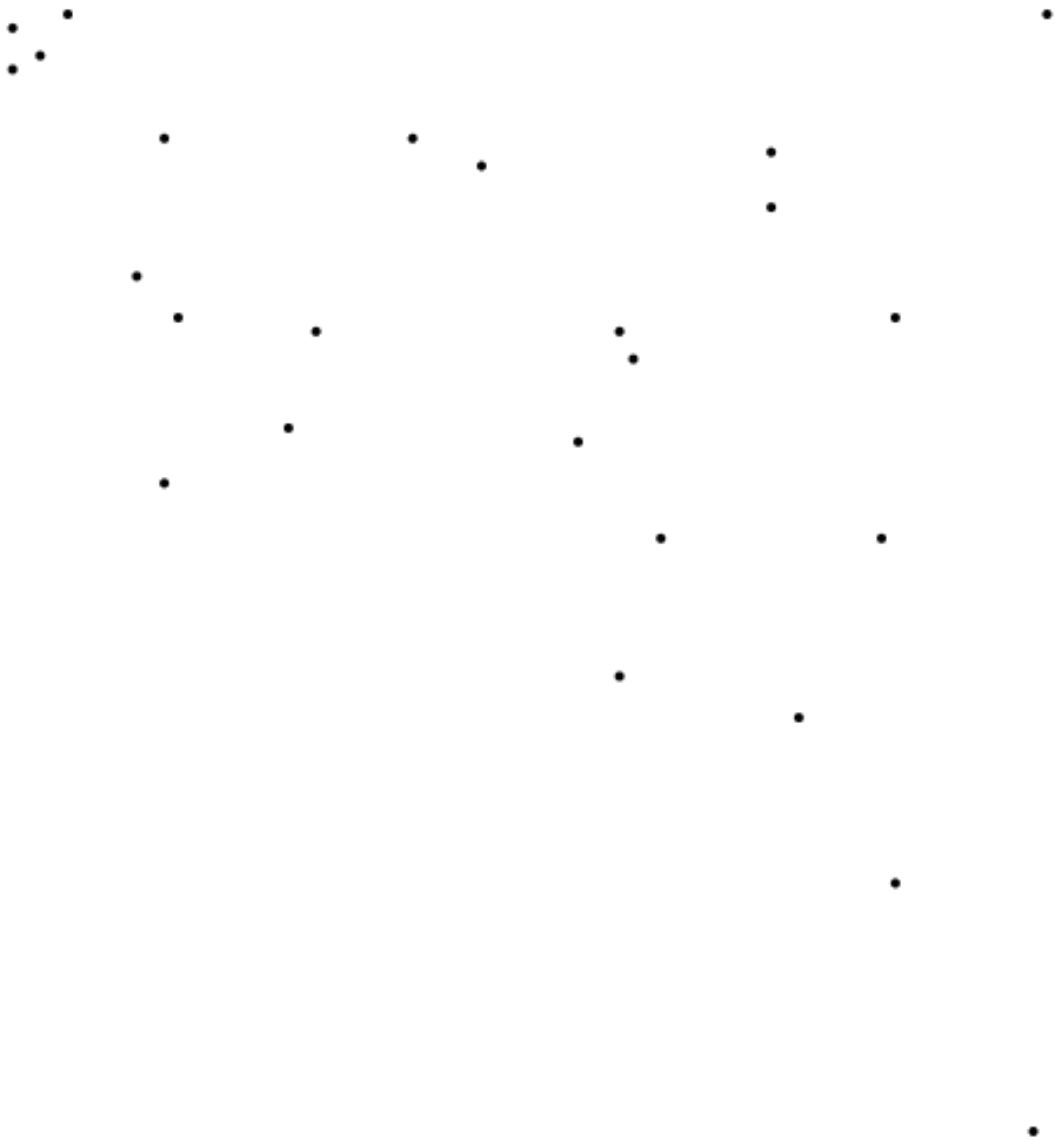
Without any constraints we would simply set every x variable to 0.

So what are the constraints? Each city should be visited right? Okay so each city needs to have a successor and a predecessor because we want a cyclic path.

$$\sum_{t=1}^{N} x_{ft} = 1 \quad \forall f \in \{1, \dots, N\} \tag{4}$$

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

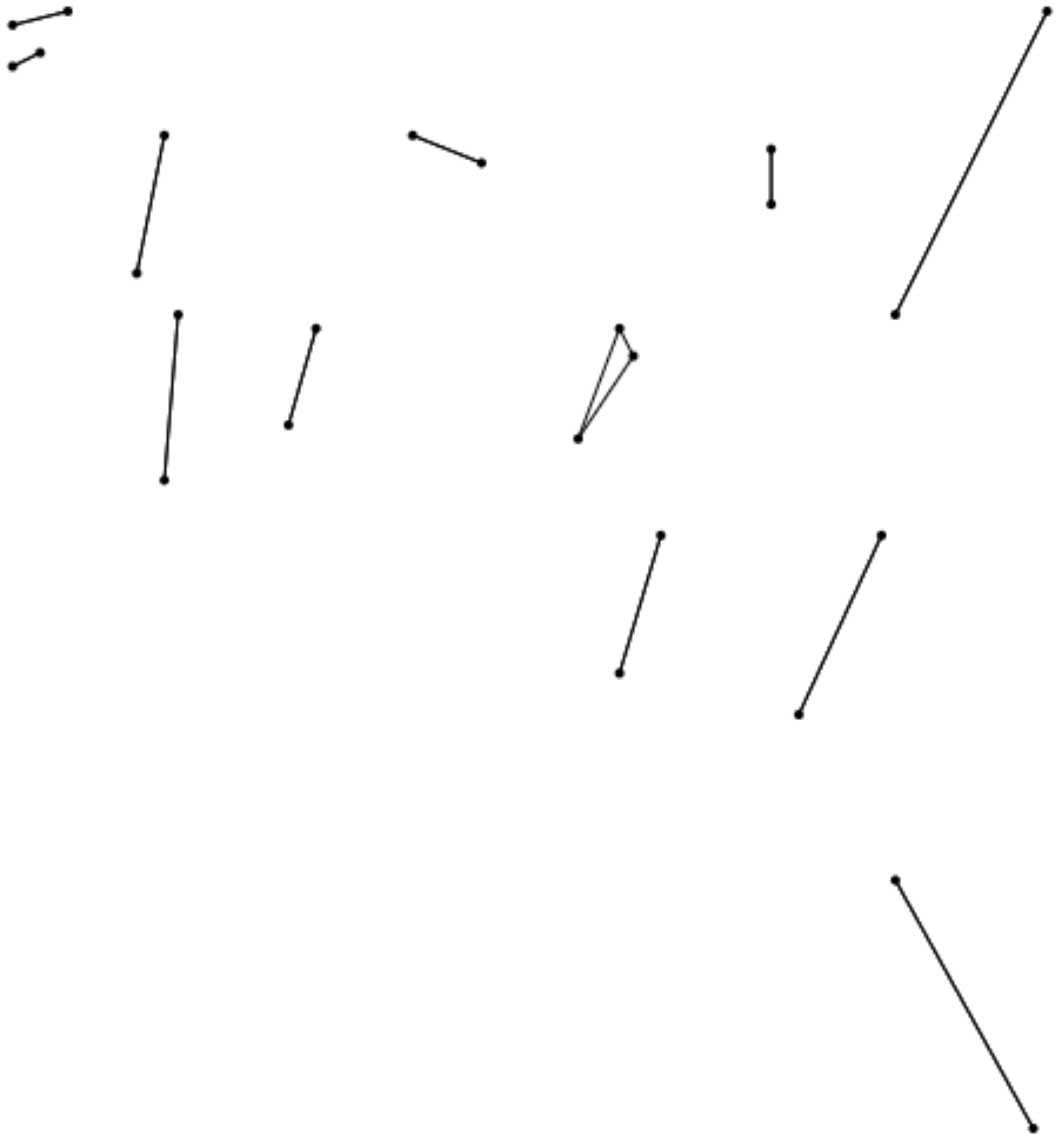
Let's run it!



What went wrong? Exactly the smart evil program connected every city to itself which clearly fulfills our constraints and is definitely optimal.

Let's add another constraint:

$$x_{ff} = 0 \quad \forall f \in \{1, \ldots, N\} \tag{5}$$

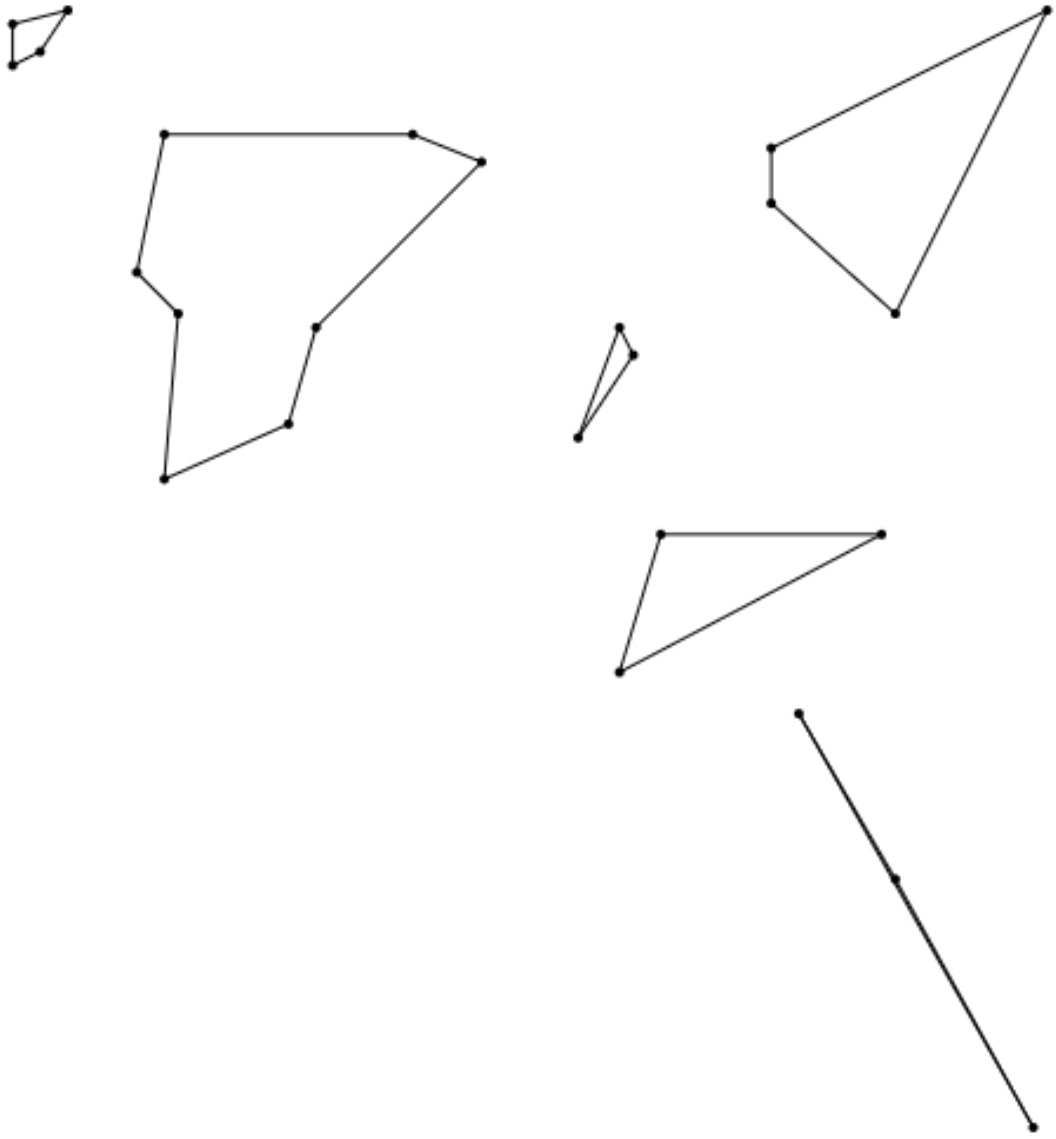


Well it might sound strange but for me this looks much better than before. It would be possible to remove the lines which just connects two points with one another by using this constraint:

$$x_{ft} + x_{tf} <= 1 \quad \forall f, t \in \{1, \ldots, N\} \tag{6}$$

This simply means that we can't connect a city f to a city t if we already connected t to f or the other way around.

This adds another 625 constraints and gives us this result:



Wow! I mean this looks awesome to me. We kind of solved six smaller TSPs. Now we have to connect them somehow. Hmm not that easy if we really want to have it optimal.

We can do the same thing as before, right? Just disallow every cycle which is smaller than the big cycle which connects every city.

It's not that easy to disallow every cycle. We would need to construct every cycle in a constraint and say that it isn't allowed.

The idea is to disallow the cycles that the program finds for you. That reduces the number of cycle constraints and we don't have to find cycles by ourself.

First of all let's right down everything we have so far in Julia.

```
m = Model(with_optimizer(GLPK.Optimizer))
@variable(m, x[1:N,1:N], Bin)
@objective(m, Min, sum(x[i,j]*euclidean(c_pos[i],c_pos[j]) for i=1:N,j=1:
for i=1:N
    @constraint(m, x[i,i] == 0)
    @constraint(m, sum(x[i,1:N]) == 1)
```

COPY

```
end
for j=1:N
    @constraint(m, sum(x[1:N,j]) == 1)
end
for f=1:N, t=1:N
    @constraint(m, x[f,t]+x[t,f] <= 1)
end

optimize!(m)
```

That's all! The first line specifies the solver we want to use. Julia or JuMP can use a lot of different open source and also commercial solvers. If we want to use `Gurobi` which is a commercial Solver we would write `m = Model(with_optimizer(Gurobi.Optimizer))`. Well and we have to use `using Gurobi` at the beginning of our file and download Gurobi etc and pay a lot of money :D Or we have an acadamic license from our university or whatever.

Whatever, JuMP makes it possible to switch the background solver that easily.

Let's specify our variables in the next row. This one line constructs N^2 in our case 625 binary variables.

```
@variable(m, x[1:N,1:N], Bin)
```

COPY

I like that short form ;)

The objective might look a bit more complicated at first.

```
@objective(m, Min, sum(x[i,j]*euclidean(c_pos[i],c_pos[j]) for i=1:N,j=1:
```

COPY

Okay first we specify that we want to minimize. Then we sum over all x variables multiplied by the distance for every possible city combination. I think the notation is pretty close to the mathematical notation.

Now our constraints:

```
for i=1:N
    @constraint(m, x[i,i] == 0) #not self
    @constraint(m, sum(x[i,1:N]) == 1) # one out
end
for j=1:N
    @constraint(m, sum(x[1:N,j]) == 1) # one in
end
for f=1:N, t=1:N
    @constraint(m, x[f,t]+x[t,f] <= 1)
end
```

COPY

Afterwards our direct loop constraints. `for f=1:N, t=1:N` is pretty neat I think because it's an easy readable form of a loop in a loop.

Our last line `optimize!(m)` solves the model. In the newer version of JuMP you can get the status by `JuMP.termination_status(m)` which in the end should be `MOI.OPTIMAL` in the end. It can be `MOI.INFEASIBLE` if we did something wrong.

MOI=MathOptInterface

Solving this gives us quite quickly the result visualized above.

We can print out the objective value using `println("Obj: ", JuMP.objective_value(m))`.

Now let's disallow the cycles.

After `optimize!(m)` we simply add a while loop which tests whether we really solved the problem.

```
while !is_tsp_solved(m)
    optimize!(m)
end
```

COPY

Now we can specify the `solved` function:

```
function is_tsp_solved(m, x)
    N = size(x)[1]
    x_val = JuMP.value.(x)

    # find cycle
    cycle_idx = 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
    println("cycle_idx: ", cycle_idx)
    println("Length: ", length(cycle_idx))
    if length(cycle_idx) < N
        @constraint(m, sum(x[cycle_idx,cycle_idx]) <= length(cycle_idx)-1
        return false
    end
    return true
end
```

COPY

First we get the current values of x. Then we find a cycle and simply start at the first city. **Remember: Our indices start with 1**.

We construct an array where we save the indices of the cities which are in our city #1 cycle. While the cycle isn't finished we find the index of the next city in our

cycle using `findmax(x_val[cycle_idx[end],1:N]). cycle_idx[end]` gives us the last index in our cycles array. `x_val[cycle_idx[end],1:N]` is the row of our `x` array which specifies to which cities the current city is connected to. Well it should be connected to only one city so if we find the maximum in this row, which is 1 and the index of it we have the next city in our cycle.

If the next index in the cycle is the first index in our cycle (which is 1) then our cycle is closed. Otherwise we continue until the path is closed.

Btw `push!` means that we push something to our array and the `!` indicates that we change the first argument (our array) using this operation.

Okay we can print out the cycle and the length of the cycle. A cycle `[1,2,3,1]` would be displayed as `[1,2,3]` with a length of 3 because we don't push the first index into the array at the end. Therefore the length is the number of edges in that path.

If the number of edges is less than the number of vertices we haven't found the big cyclic path yet. Therefore we have to disallow the cycle we constructed and solve the problem again.

```
@constraint(m, sum(x[cycle_idx,cycle_idx]) <= length(cycle_idx)-1)
```
COPY

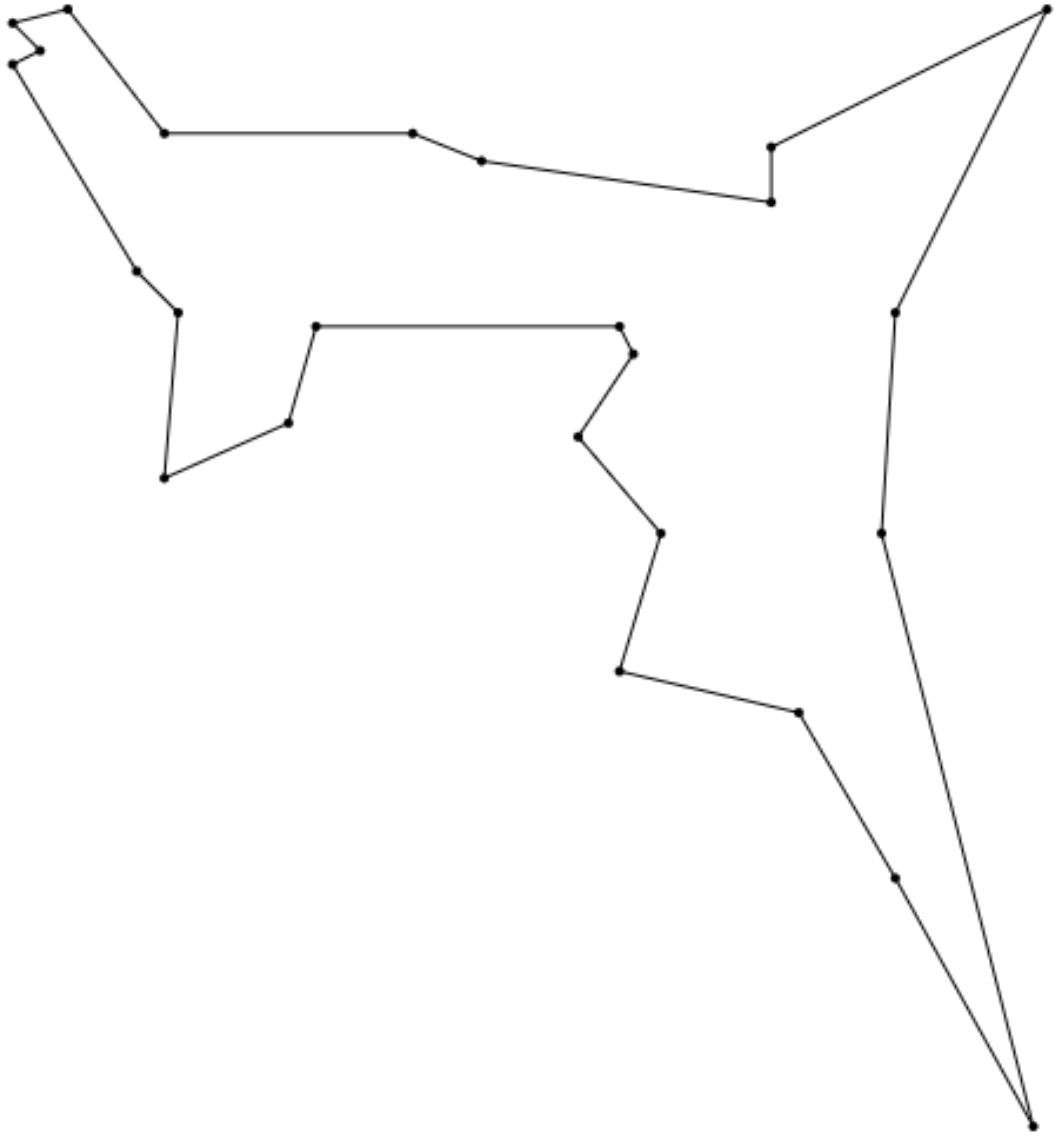
What exactly is going on here? Okay `sum(x[cycle_idx,cycle_idx])` calculates the number of connections between the cities in our cycle. That number of connections is currently the same as the length of our cycle. If we say that the number of connections in this cycle should be smaller than that we disallow this cycle but don't disallow the big cyclic path we are looking for.

Let's run it!

This is the output:

```
cycle_idx: [1, 3, 2, 4]
Length: 4
cycle_idx: [1, 3, 7, 2, 4]
Length: 5
cycle_idx: [1, 3, 7, 6, 14, 12, 8, 9, 25, 5, 4, 2]
Length: 12
cycle_idx: [1, 3, 7, 6, 14, 17, 19, 11, 12, 8, 9, 25, 5, 4, 2]
Length: 15
cycle_idx: [1, 2, 4, 5, 7, 3]
Length: 6
cycle_idx: [1, 2, 4, 5, 25, 7, 3]
Length: 7
cycle_idx: [1, 2, 4, 5, 25, 9, 8, 12, 7, 3]
Length: 10
cycle_idx: [1, 2, 4, 5, 25, 9, 8, 12, 17, 19, 11, 23, 24, 21, 18, 10, 13,
Length: 22
cycle_idx: [1, 2, 4, 5, 25, 9, 8, 12, 17, 19, 11, 23, 24, 22, 16, 20, 21,
Length: 25
```
COPY

We only have to disallow nine cycles to get this wonderful optimal solution:



Let's do a simple benchmark run: We have to put everything in a function and should remove every print statement. I normally use `solve_tsp(;benchmark=false)` and then print only if benchmark is `false`. You can see it in the GitHub Repo

```
using BenchmarkTools
```

COPY

```
@benchmark solve_tsp(;benchmark=true)
BenchmarkTools.Trial:
  memory estimate:  5.40 MiB
  allocs estimate:  149316
  --------------
  minimum time:     105.857 ms (0.00% GC)
  median time:      128.620 ms (0.00% GC)
  mean time:        137.117 ms (1.67% GC)
  maximum time:     208.441 ms (3.48% GC)
  --------------
```

```
samples:          37
evals/sample:     1
```

This includes the reading time of the file.

I think it's amazing what kind of problems are solveable with Mixed Integer Programming.

It's possible to optimize the TSP problem in different ways as well and for bigger instances if it's not really possible to definitely find the optimal solution we have to use different options but for this relatively small example it's easily solveable.

Thanks for reading ;)

You can download the Julia code and my JS code which I used for the visualization on GitHub.

If you've enjoyed this and some other posts on my blog I would appreciate a small donation either via PayPal or Patreon whereas the latter is a monthly donation you can choose PayPal for a one time thing. For the monthly donation you'll be able to read a new post earlier than everyone else and it's easier to stay in contact.

Evan Fields • 3 years ago

Great post! In a shameless plug I'll also note that in this small example, the heuristic package TravelingSalesmanHeuristics.jl also finds the same path in about .0001 seconds. :) Of course in general heuristics won't find the optimum path, and TravelingSalesmanHeuristics doesn't implement the most advanced heuristic methods.

https://github.com/evanfiel...

OpenSources Evan Fields • 3 years ago

Hey, thanks for reading. Heuristics are awesome and can also help to solve the problem to an optimal solution faster

---