Least Path Grade Transitions

Polypropylene has many potenital grades each with unique charatersitics. Each grade is is created by a certain reaction environment. Chgangeing this environment takes time and causes `off-spec` or `wide-spec` product to be produced.

To minimize this transition, it is best to move to similar reactions instead of very different reactions. If we look at a simple example, a planner would want to go from low melt flow grades to high ones and then back down in a sin wave. However, there are more complex atributes like if a grade is random of homo-polymer etc. To solve this we write a simple linear program. This is a real life application of the classic **Traveling Salesperson Problem** (TSP).

I would like to thank Evan Fields and **this** blog post for getting me started.

Loading Libraries

```
Plots.GRBackend()
```

```
begin
    using MultivariateStats
    using JuMP
    using Cbc
    using Distances
    using Distributions
    using Random
    using Plots
    gr()
end
```

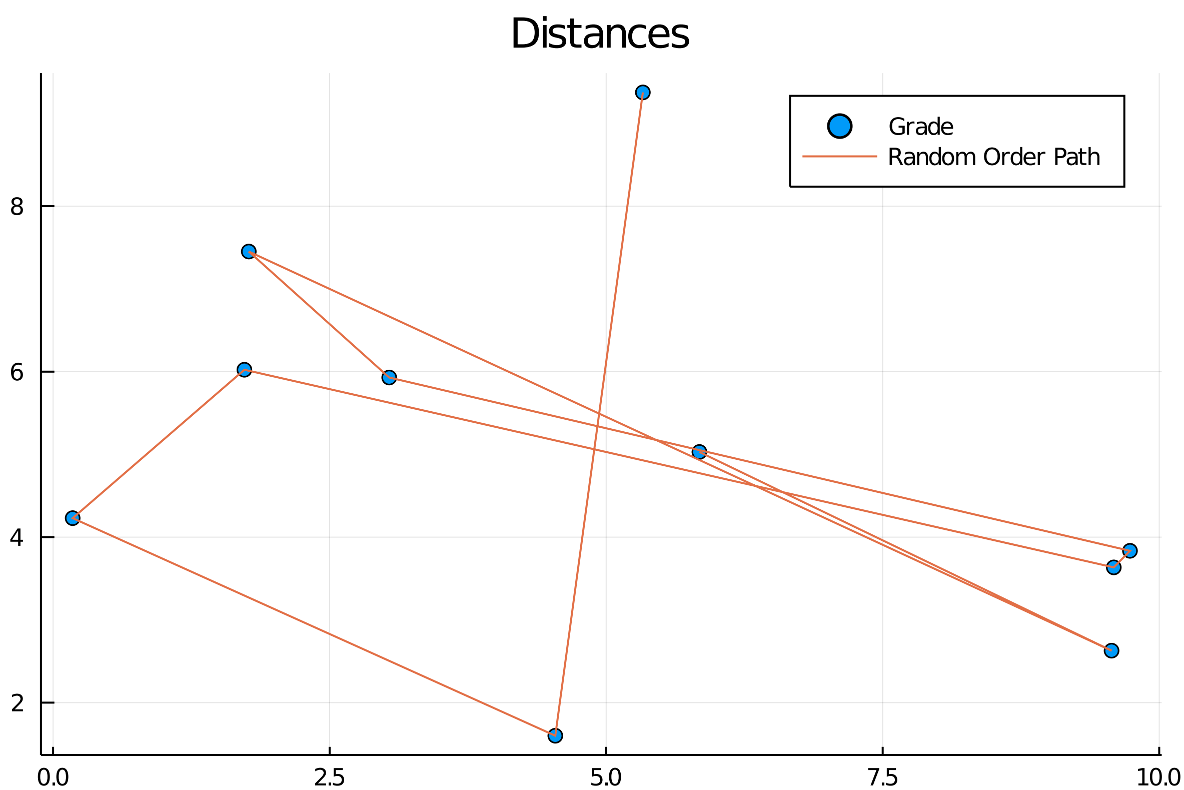
Generating Data

First we need to set the transition costs. In the real world, we multi dimensial output space to calculate the transition matrix. In order to visualize this, we are going to use a 2D vector space to more easily visualize.

This difference is a bit accademic as multi factored reasoned for costs are projected down onto a matrix.

```
Float64[9.37466, 1.60006, 4.22956, 6.02298, 3.63458, 3.83491, 5.92912, 7.45181, 2.6
```

```
begin
    len = 10
    Random.seed!(42)
    x_low_high = rand(Uniform(0.0, 10.0), len)
    y_low_high = rand(Uniform(0.0, 10.0), len)


end
```




```
begin
    plot(x_low_high, y_low_high, seriestype = :scatter, title = "Distances",
         lab = "Grade")
    plot!(x_low_high, y_low_high, lab = "Random Order Path")
end
```

This is obviously not the best paht through the production wheel.

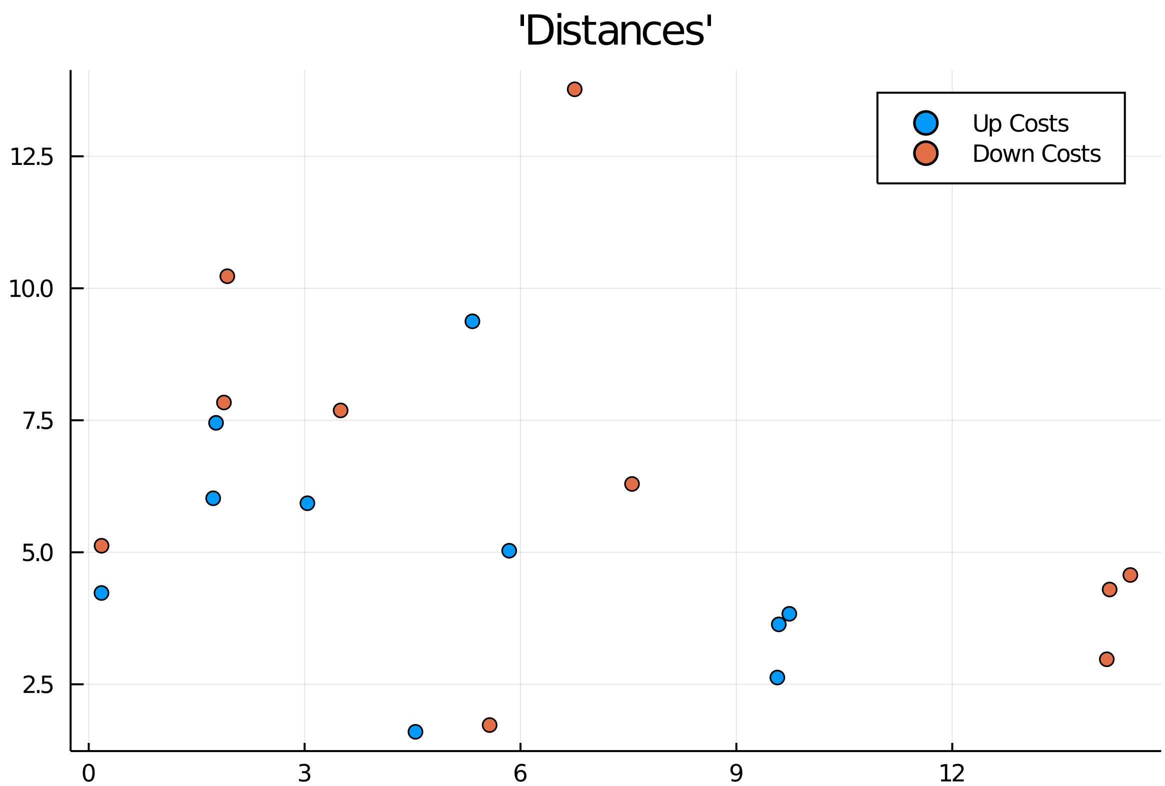
Randomizing It

It's also not the only problem we face. In polypropylen reactions, the transition from high to low is different from the transition from low to high even for the same two grades. To simulate this we add some random noise.

```
Float64[13.7689, 1.72807, 5.12402, 7.83679, 4.29509, 4.57024, 7.68684, 10.2283, 2.9
```

```
begin
    Random.seed!(42)
```

```
    N = length(x_low_high)
    x_perterbation = rand(Uniform(1.0, 1.5), N)
    y_perterbation = rand(Uniform(1.0, 1.5), N)

    x_high_low = x_low_high .* x_perterbation
    y_high_low = y_low_high .* y_perterbation
end
```




```
begin
    plot(x_low_high, y_low_high, seriestype = :scatter, title = "'Distances'", lab =
"Up Costs")
    plot!(x_high_low, y_high_low, seriestype = :scatter, lab = "Down Costs")
end
```

These perturbations exibit the increased time of transitions when going in a different direction.

Product Wheel

Our product wheel should go from low to high and then back down. If our costs were the same on the way up and down, we could just use the same optimization but we are not guaranteeded that it will be optimal.

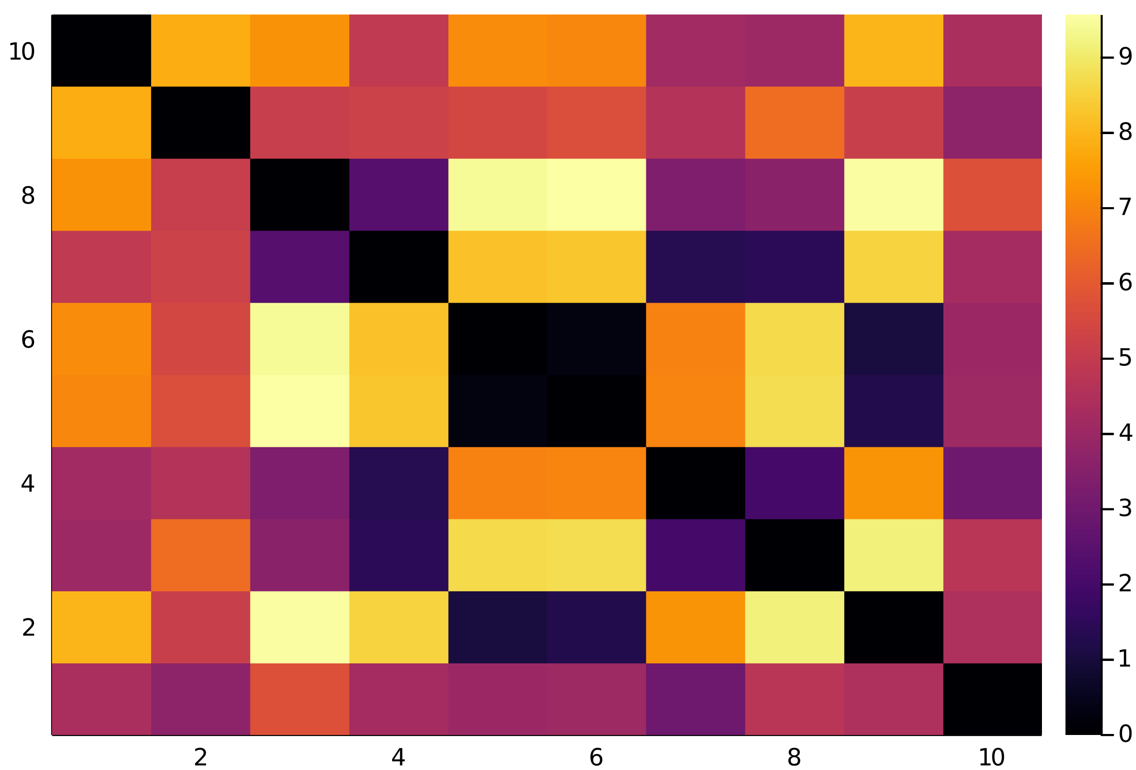
Solving asymetric problems like this is difficult so I am just going to break them up into two. If you had a traditional cost matrix that was asymetric, you need to split it into two symetric ones, one with the up costs on both sides of the diagonal and one with down costs. Since ours are artificial, I'm just going to keep them in their seperate forms. Just remember that most of the time we would need to split them.

So while we technically have a 'asymetric' problem with different values on the lower and upper triagnles of the cost matrix, we split them into two and solve each individually.

```
function cost_generator(d1, d2)
    N = length(d1)
    cost = zeros(N, N)
    points = [[d1[i], d2[i]] for i ∈ 1:N]
    for i ∈ 1:N, j ∈ 1:N
        @inbounds cost[i, j] = euclidean(points[i], points[j])
    end
    return cost
end;
```

```
cost_up =
10×10 Array{Float64,2}:
 0.0      7.81479  7.28325  4.92055  7.14662   …  4.13887  4.04852  7.96688  4.37509
 7.81479  0.0      5.09448  5.24058  5.44347      4.58209  6.47477  5.13287  3.66849
 7.28325  5.09448  0.0      2.37202  9.43118      3.32844  3.59417  9.52785  5.72217
 4.92055  5.24058  2.37202  0.0      8.2148       1.31273  1.42939  8.54331  4.23178
 7.14662  5.44347  9.43118  8.2148   0.0          6.9408   8.70209  1.0067   3.99769
 7.0769   5.65565  9.56693  8.29994  0.248122  …  7.01677  8.74919  1.21826  4.07199
 4.13887  4.58209  3.32844  1.31273  6.9408       0.0      1.98255  7.31736  2.94491
 4.04852  6.47477  3.59417  1.42939  8.70209      1.98255  0.0      9.17112  4.73951
 7.96688  5.13287  9.52785  8.54331  1.0067       7.31736  9.17112  0.0      4.43309
 4.37509  3.66849  5.72217  4.23178  3.99769      2.94491  4.73951  4.43309  0.0
```

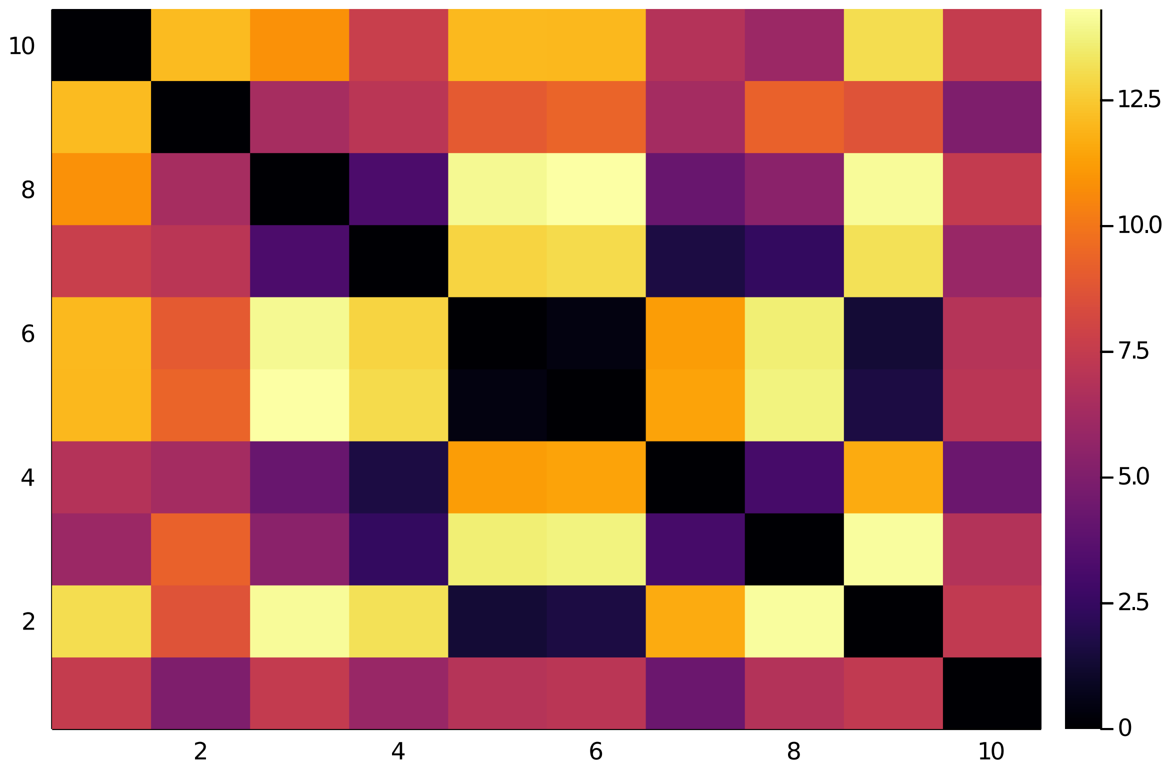
```
cost_up = cost_generator(x_low_high, y_low_high)
```




```
heatmap(cost_up[end:-1:1, :]) # make it reverse b/c plots is annoying
```

```
cost_down =
10×10 Array{Float64,2}:
  0.0       12.0987   10.861     7.67785  …   6.89728   5.98684  13.085     7.51687
 12.0987     0.0       6.37278   7.13781      6.30828   9.24894   8.66654   4.97657
 10.861      6.37278   0.0       3.20165      4.19563   5.395    14.1337    7.46367
  7.67785    7.13781   3.20165   0.0          1.62844   2.39196  13.1975    5.87695
 12.0421     8.99023  14.033    12.8075      11.2119   13.6215    1.32225   6.93174
 12.0099     9.34643  14.3071   13.0126   …  11.4084   13.7658    1.62999   7.13642
  6.89728    6.30828   4.19563   1.62844      0.0       2.98982  11.6439    4.28214
  5.98684    9.24894   5.395     2.39196      2.98982   0.0      14.2131    6.86351
 13.085      8.66654  14.1337   13.1975      11.6439   14.2131    0.0       7.38644
  7.51687    4.97657   7.46367   5.87695      4.28214   6.86351   7.38644   0.0
```

```
cost_down = cost_generator(x_high_low, y_high_low)
```




```
heatmap(cost_down[end:-1:1, :]) # make it reverse b/c plots is annoying
```

Classic Traveling Salesperson (TSP) problems are designed work for round trips. While we kind of want to a round trip with different costs assoiated with the up and down portions of the product wheel, we don't want to make our constraints too complex.

As a work around I just calculate one single up trip and one down trip. We still need to change the trips to be one way. To do this we add a trip that has astronomically high costs except for the start and end nodes which are zero. This way the end nodes are always 'next to each other' because they have the cheapest transition. The downside of this is that we have to pick the start and end nodes. To do this I picked the ones with the greatest cost (the maximum value of the cost matrix) to be the start and end.

We can then ignore this final transition when it comes to our analysis and the trip will look like it is one way.

```
"""
    make_matrix_one_way(A::AbstractMatrix)
Takes a Matrix, calculates the max possible transition cost and pads it with a vector
of that cost. The highest transition cost values (farthest cities) are set to zero to
allow for the MIP to complete a tour while maintaining uni-directional logic.
"""
function make_matrix_one_way(A::AbstractMatrix; idx = nothing)
    max_possible = sum(A)
    if idx == nothing
        idx = argmax(A)
    end
    id1, id2 = idx[1], idx[2]

    shortcut_vec = fill(max_possible, size(A, 1) + 1)
    shortcut_vec[[id1, id2, end]] .= 0.0
    cost = hcat(A, shortcut_vec[1:end-1])
    cost = vcat(cost, shortcut_vec')

    return cost, idx
end;

```

```
(11×11 Array{Float64,2}:                                                       , CartesianI
    0.0       7.81479  7.28325   4.92055  …   7.96688   4.37509  488.357
    7.81479   0.0      5.09448   5.24058      5.13287   3.66849  488.357
    7.28325   5.09448  0.0       2.37202      9.52785   5.72217    0.0
    4.92055   5.24058  2.37202   0.0          8.54331   4.23178  488.357
    7.14662   5.44347  9.43118   8.2148       1.0067    3.99769  488.357
    7.0769    5.65565  9.56693   8.29994  …   1.21826   4.07199    0.0
    4.13887   4.58209  3.32844   1.31273      7.31736   2.94491  488.357
    4.04852   6.47477  3.59417   1.42939      9.17112   4.73951  488.357
    7.96688   5.13287  9.52785   8.54331      0.0       4.43309  488.357
    4.37509   3.66849  5.72217   4.23178      4.43309   0.0      488.357
  488.357   488.357    0.0     488.357    …  488.357   488.357     0.0
```

```
cost1, _ = make_matrix_one_way(cost_up)
```

This allows us to simulate a one way trip if we remove the final index.

TSP Solution

I am using the **Dantzig–Fulkerson–Johnson formulation** to solve my problem.

It is easiest understood as thining of it in two parts. First, I wrote the `dfj` function to initialize the model `m` the `Transition` variable, and the objective function.

Secondly, the `dfj` function adds constraints to make sure that there are no zero cost joins back to themselves and then to make sure that each row and column have an exact value of 1. This makes it so each transition will happen once and only once.

While these constraints seem complete, there is actually one aditional step that, most of the time, needs to be taken.

```
"
    dfj(A::AbstractArray, optimizer)
Takes cost matrix and computes the first step of the optimization.
"
function dfj(A::AbstractArray, optimizer)
    N, M = size(A)

    m = Model(optimizer)
    @assert N == M "Matrix must be square"
    @variable(m, Transition[1:N, 1:M], Bin)

    @objective(m, Min, sum(Transition .* A))

    trans_iterator = 1:N
    for i ∈ trans_iterator
        @constraint(m, Transition[i, i] == 0)
        # ^ No self linking
        @constraint(m, sum(Transition[i, :]) == 1)
        # ^ All must be used
    end

    for j ∈ trans_iterator
        @constraint(m, sum(Transition[:, j]) == 1)
    end

    for i ∈ trans_iterator, j ∈ trans_iterator
        @constraint(m, Transition[i, j] + Transition[j, i] ≤ 1)
        # ^ makes it impossible to have to "same" tranistions
        # on the diagonal or thought of another way, cities self joining with
        # eachother
    end

    optimize!(m)
    return m
end;
```

Unfortunetly, the constraints in the `djf` function cause it to often return sub tours instead of a tour that hits every path. Think:

A -> B -> C -> A and D -> E -> D

This satisfies mini TSP problems but not the global one. We could exhaustivly limit these issues but MIPs get very difficult to solve if we have too many constraints.

A way around this is to **Delayed Column Generation** We run the model, check if it has tours that are too small, limit those tours with a new constraint, and then run it again. We repeate this until we get a satisfactory result.

This provides the optimal solution with mimimal constraints.

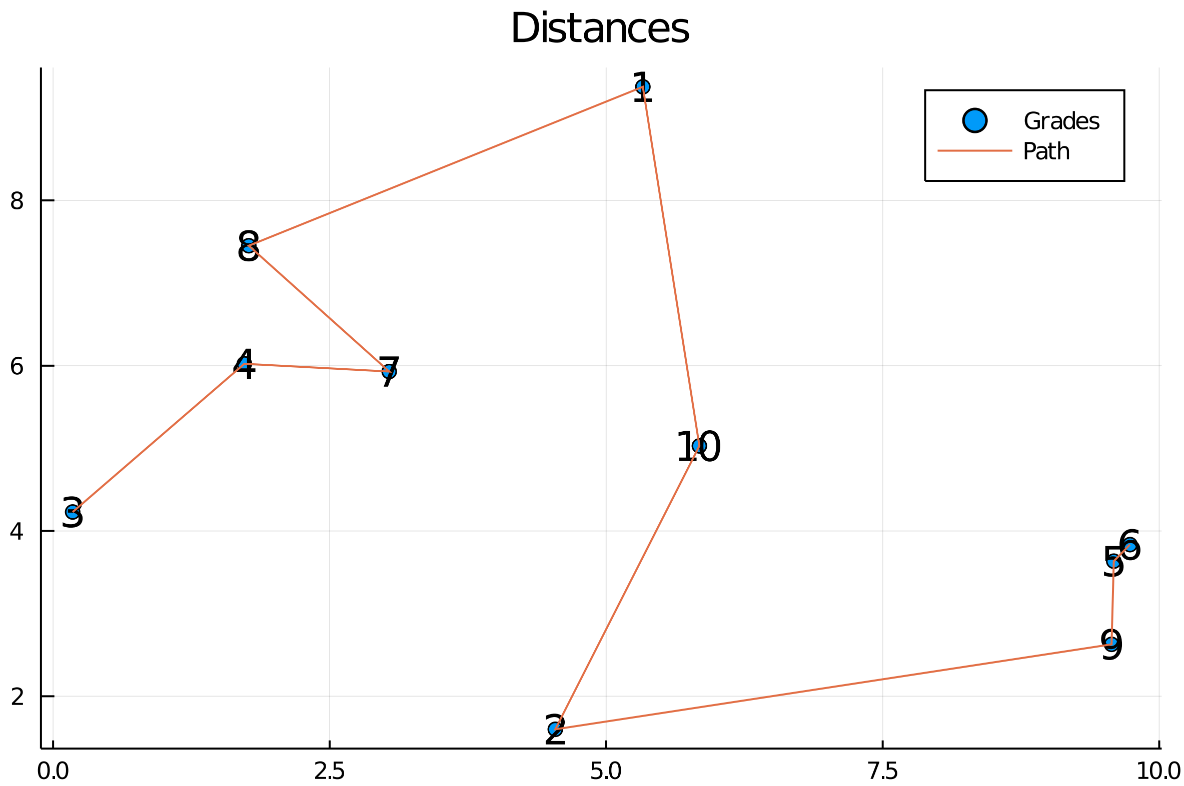
```
"   delayed_column_gen(m)
Takes a JuMP model (genereated from the dfj() and adds constraints as necessary to add complete tours.
```

```julia
"
function delayed_column_gen(m)
    var = m.obj_dict[:Transition]
    vals = value.(var)

    N = size(vals, 1)
    Q = [1]
    while true
        idx = argmax(vals[Q[end], :])
        if idx == Q[1] # checks if tour is completed and short
            break
        else
            push!(Q, idx)
        end
    end

    if length(Q) < N
        @show Q
        @constraint(m, sum(var[Q, Q]) ≤ length(Q) - 1) # 'Bans' the current solution
        optimize!(m)
        delayed_column_gen(m)
    else
        return Q
    end

end;
```

```julia
m = dfj(cost1, Cbc.Optimizer);
```

```
Int64[6, 5, 9, 2, 10, 1, 8, 7, 4, 3]
```

```julia
begin
    best_order_with_skip = delayed_column_gen(m)
    skip_idx = argmax(best_order_with_skip)
    best_order =
        vcat(best_order_with_skip[1+skip_idx:end], best_order_with_skip[1:skip_idx-1])
end
```




```
begin
    plot(
        x_low_high,
        y_low_high,
        seriestype = :scatter,
        title = "Distances",
        series_annotations = [string(i) for i ∈ 1:10],
        lab = "Grades"
    )
    plot!(x_low_high[best_order], y_low_high[best_order], lab= "Path")
end
```

Next, lets put these into convenient functions to find the up and down tours.

```
begin
    function find_path(costs, optimizer)
        costs, idx = make_matrix_one_way(costs)
        m = dfj(costs, optimizer)

        best_order = delayed_column_gen(m)
        skip_idx = argmax(best_order)
        if skip_idx == length(best_order)
            best_order = best_order[1:end-1]
        elseif skip_idx == 1
            best_order = best_order[2:end]
        else
            best_order = vcat(best_order_with_skip[1+skip_idx:end],
                              best_order_with_skip[1:skip_idx-1])
        end
        return best_order, idx

    end
    function find_path(costs, optimizer, start_end)

        costs, idx = make_matrix_one_way(costs, idx=start_end)
        m = dfj(costs, optimizer)
```

```
        best_order = delayed_column_gen(m)
        # return best_order
        skip_idx = argmax(best_order)
        if skip_idx == length(best_order)
            best_order = best_order[1:end-1]
        elseif skip_idx == 1
            best_order = best_order[2:end]
        else
            best_order = vcat(best_order_with_skip[1+skip_idx:end],
                              best_order_with_skip[1:skip_idx-1])
        end
        return best_order
    end
end;
```

```
(Int64[6, 5, 9, 2, 10, 1, 8, 7, 4, 3], CartesianIndex(6, 3))
```

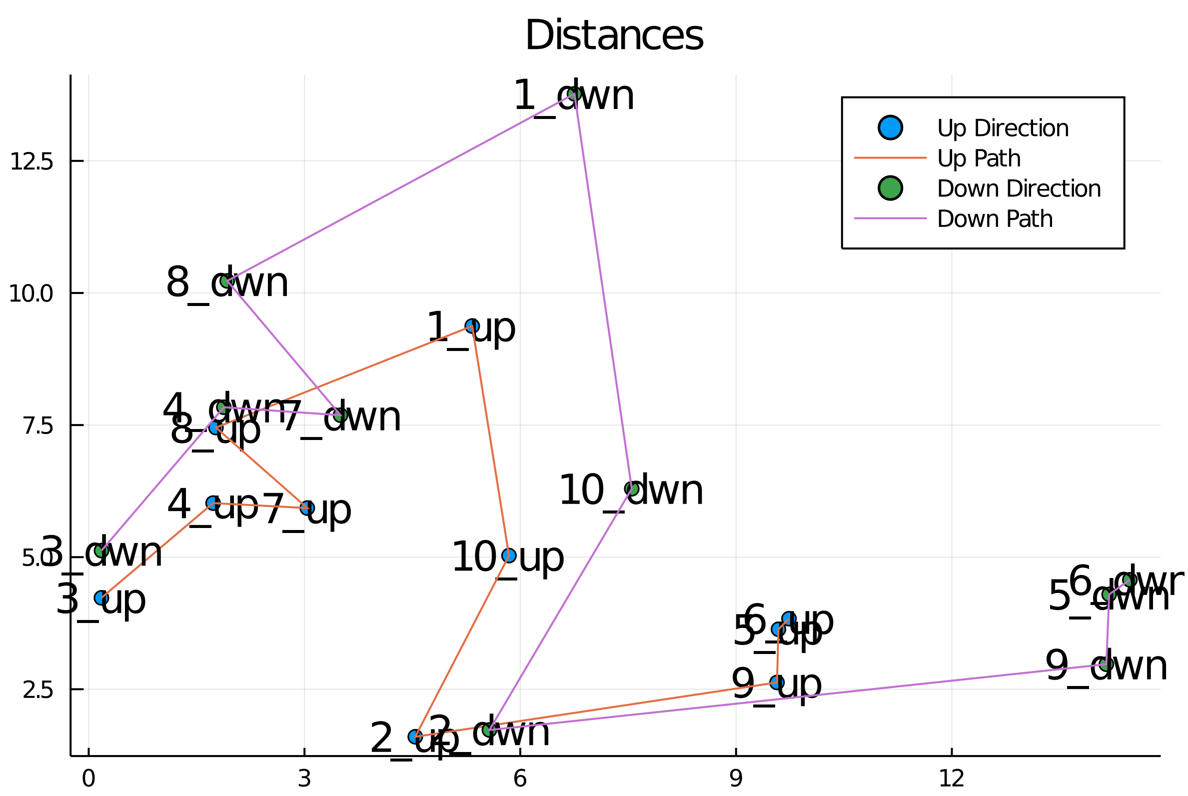
```
up_order, idx = find_path(cost_up, Cbc.Optimizer)
```

Note in the bellow function call we are using `idx` which substitutes the maximum cost of the down matrix with the start and end nodes of the previous up path.

If instead we wanted to arbitrarily set the start and end points we could supply them directly for both the up and down paths.

```
down_order =  Int64[6, 5, 9, 2, 10, 1, 8, 7, 4, 3]
```

```
down_order = find_path(cost_down, Cbc.Optimizer, idx)
```




```
begin
    plot(
        x_low_high,
```

```
        y_low_high,
        seriestype = :scatter,
        title = "Distances",
        series_annotations = [string(i) * "_up" for i ∈ 1:10],
        lab = "Up Direction"
    )
    plot!(x_low_high[up_order], y_low_high[up_order], lab = "Up Path")

    plot!(
        x_high_low,
        y_high_low,
        seriestype = :scatter,
        title = "Distances",
        series_annotations = [string(i) * "_dwn" for i ∈ 1:10],
        lab = "Down Direction"
    )
    plot!(x_high_low[down_order], y_high_low[down_order], lab = "Down Path")

end
```

```
(
 1:   Int64[6, 5, 9, 2, 10, 1, 8, 7, 4, 3]
 2:   Int64[6, 5, 9, 2, 10, 1, 8, 7, 4, 3]
)
```

```
(up_order, down_order)
```

With a seed of 42, we get the same values for up and down. This will often be the case but they can potentially be different.

Final Steps

Often we will just have a cost matrix with no corresponding x, y vectors that we can plot the path on. (Remember above we calculated the costs with the Euclideian function.)

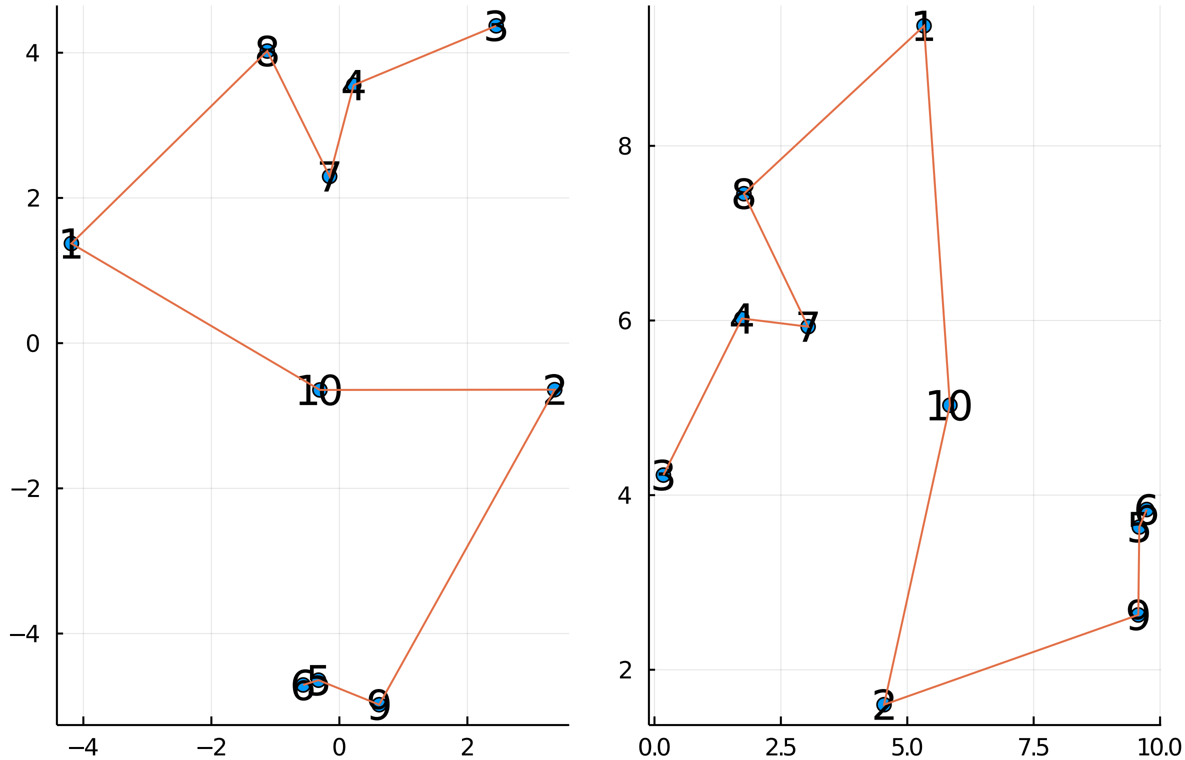
We use a technique called **Classical Multidimensional Scaling (MDS)** to project the distance matrix into two vectors.

The orientation and scaling are different but reletive positions are maintained.

This captures the non-linear transform of the Euclidean formula very nicely. In real world grade transitions it may not perform exactly as well but it should at least maintain rough differences.

```
est =
2×10 Array{Float64,2}:
  1.37212  -0.64112  4.37075  3.55408   -4.64228  …   4.02294  -4.97994   -0.644655
 -4.18653   3.36448  2.45078  0.223783  -0.32633     -1.12652   0.622047  -0.304009
```

```
est = classical_mds(cost_up, 2)
```




```
begin
    d1, d2 = (2, 1)
    l = @layout [a b]
    p1 = plot(est[d1, :], est[d2, :],
              seriestype = :scatter,
              series_annotations = [string(i) for i ∈ 1:10])
    plot!(est[d1, :][up_order], est[d2, :][up_order])
    p2 = plot(x_low_high, y_low_high,
                seriestype = :scatter,
        series_annotations = [string(i) for i ∈ 1:10])
    plot!(x_low_high[up_order], y_low_high[up_order])

    plot(p1, p2, layout = l, legend = false)
end

```