2021/12/05

Traveling Salesman Project - Phase 5

Course: MTH6412B

Under the supervision of Prof. Dominique Orban

Names:

- 1]Mozhgan Moeintaghavi
- 2]Farhad Rahbarnia

[1] Mathematics and Industrial engineering Faculty, Polytechnique Montréal, **Email** [2]Mathematics and Industrial engineering Faculty, Polytechnique Montréal, **Email**

Github repository URL:

Code to Github Phase5: https://github.com/farhadrclass/mth6412b-starter-code/tree/phase-5

Introduction

The project's final phase comprises applying the TSP solution algorithm to reconstruct jagged images. The shredder.zip file has been downloaded.

Splitting each image into a number of vertical bands and rearranging those bands in a random sequence resulted in the jagged images. In this implementation, the bands are all the same width.

The following sections go through the modifications we have made to the files read_stsp.jl, and graph.jl file in the previous phases of the project , and the creation of mainImg.jl file.

An overview of what has been accomplished

In this phase of the project, we reconstructed the provided images as accurately as possible.

After imagining that each vertical band represents a node of the complete graph, and that the weight of each edge is a measure of dissimilarity between two vertical bands, we could achieve this using the TSP solution algorithm.

We suppose that two adjacent bands in the original image are quite similar, with only a few differences, and then we seek a straightforward path through the graph's nodes that simultaneously has maximum length and minimum weight. We know that this problem is the same as TSP, as we saw in class.

We have also installed the following four modules:

- FileIO
- Pictures
- ImageView
- ImageMagick

We used the created tsp file for each shreded image. We updated the read_stsp.jl file, which reads edge weights, in order to read these files. The data field of the nodes is the index of the node.

Also, we read a jagged image and determined the dissimilarity between each pair of columns using the provided command.

The codes and the related explanations are imported in the following sections of the report.

The mainImg.jl file

This file is the primary image reconstruction file. Here, we either read the graph from the provided tsp file or create it using data types. We read the graphs in the tsp instances and use the createGraph() function, and then name it based on its assigned name or we assign a name for it. Alternatively, we create it using data types. A flag is added if the nodes are read, you will get an edge list, which is then used to obtain the node and assign to it.

Also, we locate the first node with the number 1 as its name. After that, we delete the first one from the list.

Finally, we delete the node root as well as all of its edges.

We start at the first node and locate the tour after eliminating the greatest edge and adding node 1 to it.

The written code is provided as below:

```
"""
This is  the main file for image reconstruction
"""

using Pkg
using Random
using FileIO
using Images, ImageView, ImageMagick
include(joinpath(@__DIR__, "shredder-julia", "bin", "tools.jl"))


# Import the other files
include("node.jl")
include("edge.jl")
include("graph.jl")
include("read_stsp.jl")

include("kruskal.jl")
include("prim.jl")
include("RSL.jl")
include("HK.jl")
include("GraphPlot.jl")


function createGraph(path, graphName)
    # read the graph from the file
    # localGraphName=string(path,graphName)
    fileName = joinpath(@__DIR__, "shredder-julia", "tsp", "instances", graphName *
".tsp")
    graph_nodes, graph_edges = read_stsp(fileName)

    if (length(graph_nodes) > 0) # check to see if the name is assigned in the TSP f
```

```
ile, if not we do something else
        nodesList = Node{typeof(graph_nodes[1])}[]
        vert1List = Node{typeof(graph_nodes[1])}[]
        vert2List = Node{typeof(graph_nodes[1])}[]
    else
        nodesList = Node{Int64}[]
        vert1List = Node{Int64}[]
        vert2List = Node{Int64}[]
    end


    for k = 1:length(graph_edges)
        if (length(graph_nodes) > 0) # check to see if the name is assigned in the T
SP file, if not we do something else
            node_buff = Node(string(k), graph_nodes[k], nothing, 0, 0, 0)
        else
            node_buff = Node(string(k), k, nothing, 0, 0, 0) #name is the same as we
assign it
        end
        push!(nodesList, node_buff)
    end

    # edge positions
    # go through the edge list and create the edges of the graph


    edgesList = Edge[]


    # add a flag if the nodes are read then you have a edge list then use it to get
 the node then assign

    for k = 1:length(graph_edges)
        for item in graph_edges[k]
            edge_buff = Edge(nodesList[k], nodesList[item[1]], item[2])
            push!(edgesList, edge_buff)
        end
    end


    # create a graph using data types
    # G = Graph(graphName, nodesList, edgesList)
    G = Graph(graphName, nodesList, Edge[], vert1List, vert2List)
    #adding the edges here so we test there is no dublicate
    for k = 1:length(edgesList)
        add_edge!(G, edgesList[k])
    end
    println("Finished creating a graph")

    # show(G)
    G
end


RSL_flag = true
println("Reading all the images \n\n\n")
for fileName in readdir(joinpath(@__DIR__, "shredder-julia", "tsp", "instances"))
    fileName = replace(fileName, ".tsp" => "") # removing tsp since createGraph expc
et only the name
    println("reading the file: ", fileName)
    BufferG = createGraph(joinpath(@__DIR__, "shredder-julia", "tsp", "instances"),
fileName)

    # find the first node that has 1 as the name
    root = nodes(BufferG)[findfirst(n -> name(n) == "1", nodes(BufferG))]
```

```
    #Remove the first one from the list
    myG = deepcopy(BufferG)
    deleteat!(edges(myG), findall(x -> (name(root) == name(node1(x)) || name(root) =
= name(node2(x))), edges(myG)))

    # remove the node root and all the edges from it
    deleteat!(nodes(myG), findall(x -> name(x) == name(root), nodes(myG)))



    if RSL_flag
        println("RSL has been selected")
        cycleWeight, Cycle = RSL(1, nodes(myG)[1], myG) # if 1 we use prim otherwise
kruskal
        println("RSL weightGraph ", cycleWeight, " graph weightGraph")

    else #HK
        Cycle_HK = HK_solver(1, root, myG, 100)
        cycleWeight_HK = weightGraph(Cycle)
    end

    # Once a tour has been identified, construct a list of nodes along that tour wit
hout removing the
    # the zero node. Using the write_tour() function, create a .tour file in TSPLib
 format that
    # describes your tour. Sample .tour files are available in the tsp/tours directo
ry
    # directory; these have been identified by a TSP solving method, but do not nece
ssarily give an
    # necessarily give an optimal solution.

    ########################################################################
    # HERE I remove the largest edge  and add the node 1 to it
    buff, idx = findmax(x -> x.weight, edges(Cycle))
    maxEdge = edges(Cycle)[idx]
    dummyNode1 = node1(maxEdge)
    dummyNode2 = node2(maxEdge)
    deleteat!(edges(Cycle), idx)

    add_node!(Cycle, root)
    DummyEdge = findall(x -> ((name(root) == name(node1(x)) && name(dummyNode2) == n
ame(node2(x))) || (name(root) == name(node2(x)) && name(dummyNode2) == name(node1(x)
)) || (name(root) == name(node1(x)) && name(dummyNode1) == name(node2(x))) || (name(
root) == name(node2(x)) && name(dummyNode1) == name(node1(x)))), edges(BufferG))
    for e in DummyEdge
        add_edge!(Cycle, edges(BufferG)[e])
    end
    ########################################################

    myTourSize = nb_nodes(Cycle) + 1
    myTour = zeros(Int, myTourSize)

    nextNode = "1" # first node
    myTour[1] = -1
    myTour[2] = 1
    # we start at the first node and find the tour
    for i = 3:myTourSize
        idx = findfirst(edge -> nextNode in [name(node1(edge)), name(node2(edge))] &
& !(string(myTour[i-2]) in [name(node1(edge)), name(node2(edge))]), edges(Cycle))
        if nextNode == name(node1(edges(Cycle)[idx]))
            nextNode = name(node2(edges(Cycle)[idx]))
        else
            nextNode = name(node1(edges(Cycle)[idx]))
        end
```

```
            myTour[i] = parse(Int, nextNode)
        end
        myTour = myTour[2:end]
        myTour = myTour .- 1 # we expct the tour to start from zero
        myTourFile = joinpath(@__DIR__, "shredder-julia", "tsp", "tours", fileName * ".t
our")


        write_tour(myTourFile, myTour, Float32(cycleWeight)) # why in write_tour it expe
ct Float32!!!

        inputFilename = joinpath(@__DIR__, "shredder-julia", "images", "shuffled", fileN
ame * ".png")
        outputFilename = joinpath(@__DIR__, "shredder-julia", "images", "ourReconst", fi
leName * "-reconstructed.png")
        reconstruct_picture(myTourFile, inputFilename, outputFilename)



        println("-----------------------------------------------------")

    end
```

The read_stsp.jl file modification

```
using Plots

"""Analyse un fichier .tsp et renvoie un dictionnaire avec les données de l'entête.
EN: Parses a .tsp file and returns a dictionary with the header data.
"""
function read_header(filename::String)

  file = open(filename, "r")
  header = Dict{String}{String}()
  sections = ["NAME", "TYPE", "COMMENT", "DIMENSION", "EDGE_WEIGHT_TYPE", "EDGE_WEIG
HT_FORMAT",
  "EDGE_DATA_FORMAT", "NODE_COORD_TYPE", "DISPLAY_DATA_TYPE"]

  # Initialize header
  for section in sections
    header[section] = "None"
  end

  for line in eachline(file)
    line = strip(line)
    data = split(line, ":")
    if length(data) >= 2
      firstword = strip(data[1])
      if firstword in sections
        header[firstword] = strip(data[2])
      end
    end
  end
  close(file)
  return header
end

"""Analyse un fichier .tsp et renvoie un dictionnaire des noeuds sous la forme {id =
> [x,y]}.
Si les coordonnées ne sont pas données, un dictionnaire vide est renvoyé.
Le nombre de noeuds est dans header["DIMENSION"].
EN:
Parses a .tsp file and returns a dictionary of nodes in the form {id => [x,y]}.
If coordinates are not given, an empty dictionary is returned.
The number of nodes is in header["DIMENSION"].

"""
function read_nodes(header::Dict{String}{String}, filename::String)

  nodes = Dict{Int}{Vector{Float64}}()
  node_coord_type = header["NODE_COORD_TYPE"]
  display_data_type = header["DISPLAY_DATA_TYPE"]


  if !(node_coord_type in ["TWOD_COORDS", "THREED_COORDS"]) && !(display_data_type i
n ["COORDS_DISPLAY", "TWOD_DISPLAY"])
    return nodes
  end

  file = open(filename, "r")
  dim = parse(Int, header["DIMENSION"])
  k = 0
```

```
  display_data_section = false
  node_coord_section = false
  flag = false

  for line in eachline(file)
    if !flag
      line = strip(line)
      if line == "DISPLAY_DATA_SECTION"
        display_data_section = true
      elseif line == "NODE_COORD_SECTION"
        node_coord_section = true
      end

      if (display_data_section || node_coord_section) && !(line in ["DISPLAY_DATA_SE
CTION", "NODE_COORD_SECTION"])
        data = split(line)
        nodes[parse(Int, data[1])] = map(x -> parse(Float64, x), data[2:end])
        k = k + 1
      end

      if k >= dim
        flag = true
      end
    end
  end
  close(file)
  return nodes
end

"""Fonction auxiliaire de read_edges, qui détermine le nombre de noeud à lire
en fonction de la structure du graphe.
EN:
Auxiliary function of read_edges, which determines the number of nodes to read accor
ding to the structure of the graph.

"""
function n_nodes_to_read(format::String, n::Int, dim::Int)
  if format == "FULL_MATRIX"
    return dim
  elseif format in ["LOWER_DIAG_ROW", "UPPER_DIAG_COL"]
    return n+1
  elseif format in ["LOWER_DIAG_COL", "UPPER_DIAG_ROW"]
    return dim-n
  elseif format in ["LOWER_ROW", "UPPER_COL"]
    return n
  elseif format in ["LOWER_COL", "UPPER_ROW"]
    return dim-n-1
  else
    error("Unknown format - function n_nodes_to_read")
  end
end

"""Analyse un fichier .tsp et renvoie l'ensemble des arêtes sous la forme d'un table
au.
EN: Parses a .tsp file and returns the set of edges as an array."""
function read_edges(header::Dict{String}{String}, filename::String)

  edges = []
  edges_weight = Dict{Vector{Float64}}{Float64}() # we save them as (1,2) weigth 34.
4
  edge_weight_format = header["EDGE_WEIGHT_FORMAT"]
  known_edge_weight_formats = ["FULL_MATRIX", "UPPER_ROW", "LOWER_ROW",
  "UPPER_DIAG_ROW", "LOWER_DIAG_ROW", "UPPER_COL", "LOWER_COL",
  "UPPER_DIAG_COL", "LOWER_DIAG_COL"]
```

```
    if !(edge_weight_format in known_edge_weight_formats)
      @warn "unknown edge weight format" edge_weight_format
      return edges
    end

    file = open(filename, "r")
    dim = parse(Int, header["DIMENSION"])
    edge_weight_section = false
    k = 0
    n_edges = 0
    i = 0
    n_to_read = n_nodes_to_read(edge_weight_format, k, dim)
    flag = false

    for line in eachline(file)
      line = strip(line)
      if !flag
        if occursin(r"^EDGE_WEIGHT_SECTION", line)
          edge_weight_section = true
          continue
        end

        if edge_weight_section
          data = split(line)
          n_data = length(data)
          start = 0

          while n_data > 0
            n_on_this_line = min(n_to_read, n_data)
            for j = start : start + n_on_this_line - 1
              weight = parse(Float64, data[j+1])
              n_edges = n_edges + 1
              if edge_weight_format in ["UPPER_ROW", "LOWER_COL"]
                edge = (k+1, i+k+2, weight)
              elseif edge_weight_format in ["UPPER_DIAG_ROW", "LOWER_DIAG_COL"]
                edge = (k+1, i+k+1, weight)
              elseif edge_weight_format in ["UPPER_COL", "LOWER_ROW"]
                edge = (i+k+2, k+1, weight)
              elseif edge_weight_format in ["UPPER_DIAG_COL", "LOWER_DIAG_ROW"]
                edge = (i+1, k+1, weight)
              elseif edge_weight_format == "FULL_MATRIX"
                edge = (k+1, i+1, weight)
              else
                warn("Unknown format - function read_edges")
              end

              # Way one
              ## weight = parse(Float64, data[j+1]) # turn string to Int64 then float6
4   # TODO change this to be a array of weights
              ## edges_weight[edge] = map(x -> parse(Float64, x)  data[j+1]) #change t
he name later  or data[j-start+1]
              # way two
              # We will include the weight in the data
              push!(edges, edge)
              i += 1
            end

            n_to_read -= n_on_this_line
            n_data -= n_on_this_line

            if n_to_read <= 0
              start += n_on_this_line
              k += 1
              i = 0
              n_to_read = n_nodes_to_read(edge_weight_format, k, dim)
```

```
            end

            if k >= dim
              n_data = 0
              flag = true
            end
          end
        end
      end
    end
    close(file)
    return edges#, edges_weight
end

"""Renvoie les noeuds et les arêtes du graphe.

En: Returns the nodes and edges of the graph."""

function read_stsp(filename::String)
    Base.print("Reading of header : ")
    header = read_header(filename)
    println("✓")
    dim = parse(Int, header["DIMENSION"])
    edge_weight_format = header["EDGE_WEIGHT_FORMAT"]

    Base.print("Reading of nodes : ")
    graph_nodes = read_nodes(header, filename)
    println("✓")

    Base.print("Reading of edges : ")
    edges_brut = read_edges(header, filename)
    graph_edges = []
    for k = 1 : dim
      edge_list =  Tuple{Int, Float64}[]#  Int[] # change this to be a tuple
      push!(graph_edges, edge_list)
    end

    # Here it adds edges to each nodes
    for edge in edges_brut
      if edge_weight_format in ["UPPER_ROW", "LOWER_COL", "UPPER_DIAG_ROW", "LOWER_DIA
G_COL"]
        push!(graph_edges[edge[1]], (edge[2],edge[3])) # I am adding weight to the edg
es
      else
        push!(graph_edges[edge[2]], (edge[1], edge[3]))
      end
    end

    for k = 1 : dim
      graph_edges[k] = sort(graph_edges[k])
    end
    println("✓")
    return graph_nodes, graph_edges
end

"""Affiche un graphe étant données un ensemble de noeuds et d'arêtes.
EN: Displays a graph given a set of nodes and edges.
Exemple :

    graph_nodes, graph_edges = read_stsp("bayg29.tsp")
    plot_graph(graph_nodes, graph_edges)
    savefig("bayg29.pdf")
"""
```

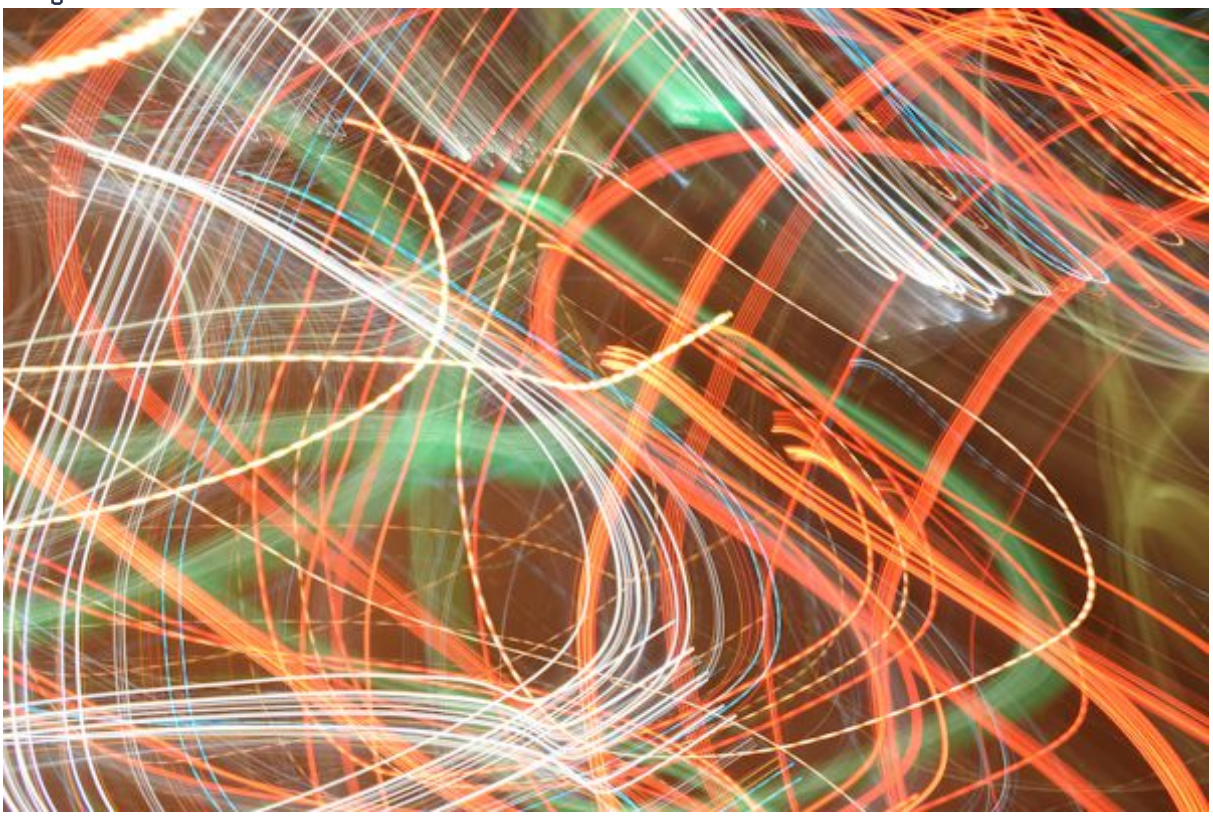
```
function plot_graph(nodes, edges)
  fig = plot(legend=false)

  # edge positions
  for k = 1 : length(edges)
    for j in edges[k]
      plot!([nodes[k][1], nodes[j[1]][1]], [nodes[k][2], nodes[j[1]][2]],
          linewidth=1.5, alpha=0.75, color=:lightgray)
    end
  end

  # node positions
  xys = values(nodes)
  x = [xy[1] for xy in xys]
  y = [xy[2] for xy in xys]
  scatter!(x, y)

  fig
end
# """Fonction de commodité qui lit un fichier stsp et trace le graphe EN: Convenienc
e function that reads an stsp file and plots the graph"""
function plot_graph(filename::String)
  graph_nodes, graph_edges = read_stsp(filename)
  plot_graph(graph_nodes, graph_edges)
end
```

Examples of an output of the system

First, we tried to implement Held and Karp algorithm, however, since the running was really slow, we considered that there might be a small bug, and instead, we tested with RSL and and also Prim algorithm for the creation of the minimum spanning tree.

Below are some examples of the reconstructed images vs orginal images:

Original =



Reconstructed1 =

Original2 =

Reconstructed2 =

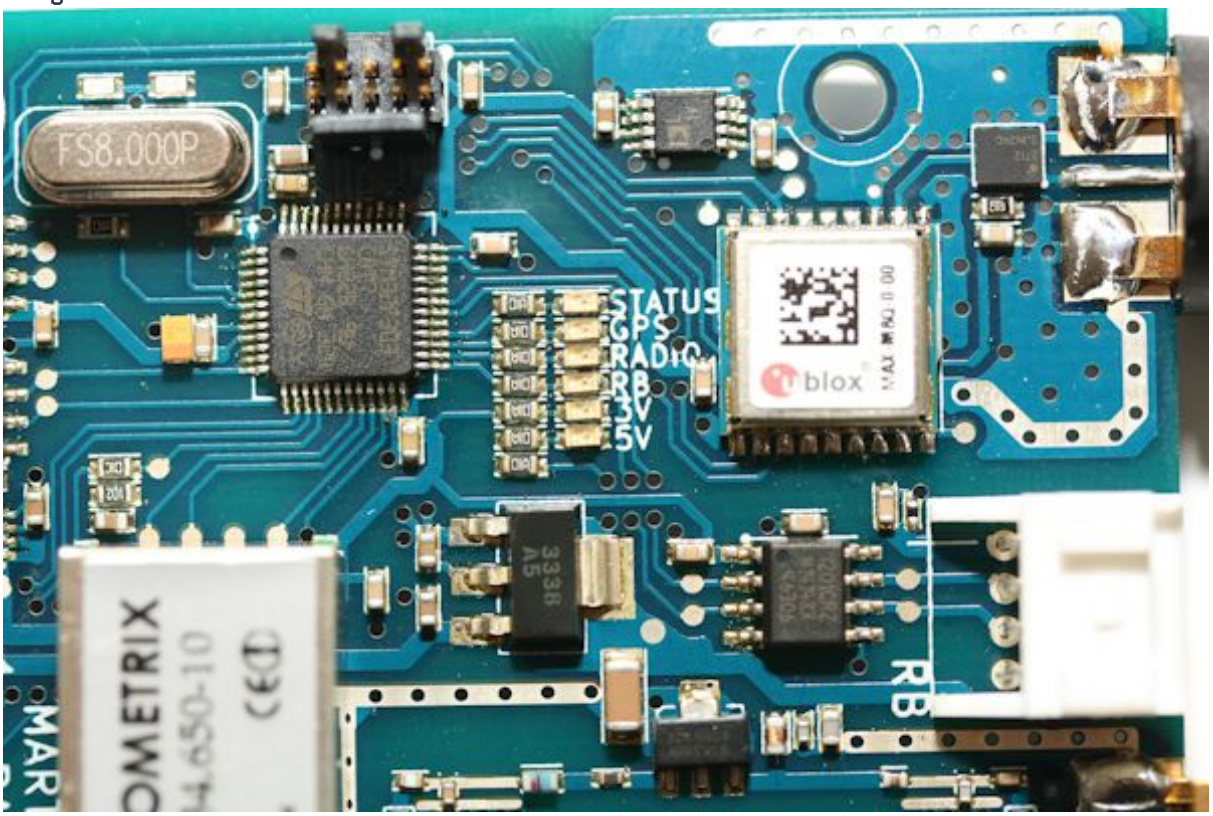
Original3 =

Reconstructed3 =

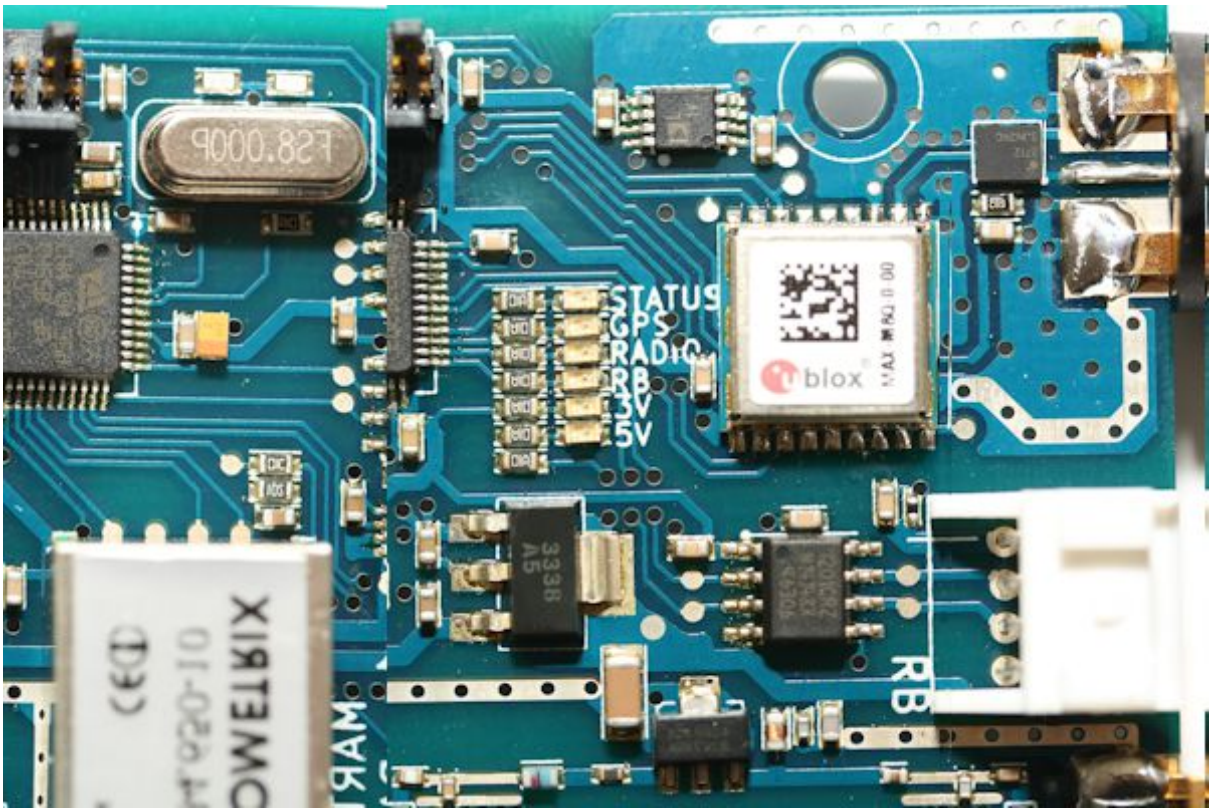
Original4 =

Reconstructed4 =

Original5 =



Reconstructed5 =



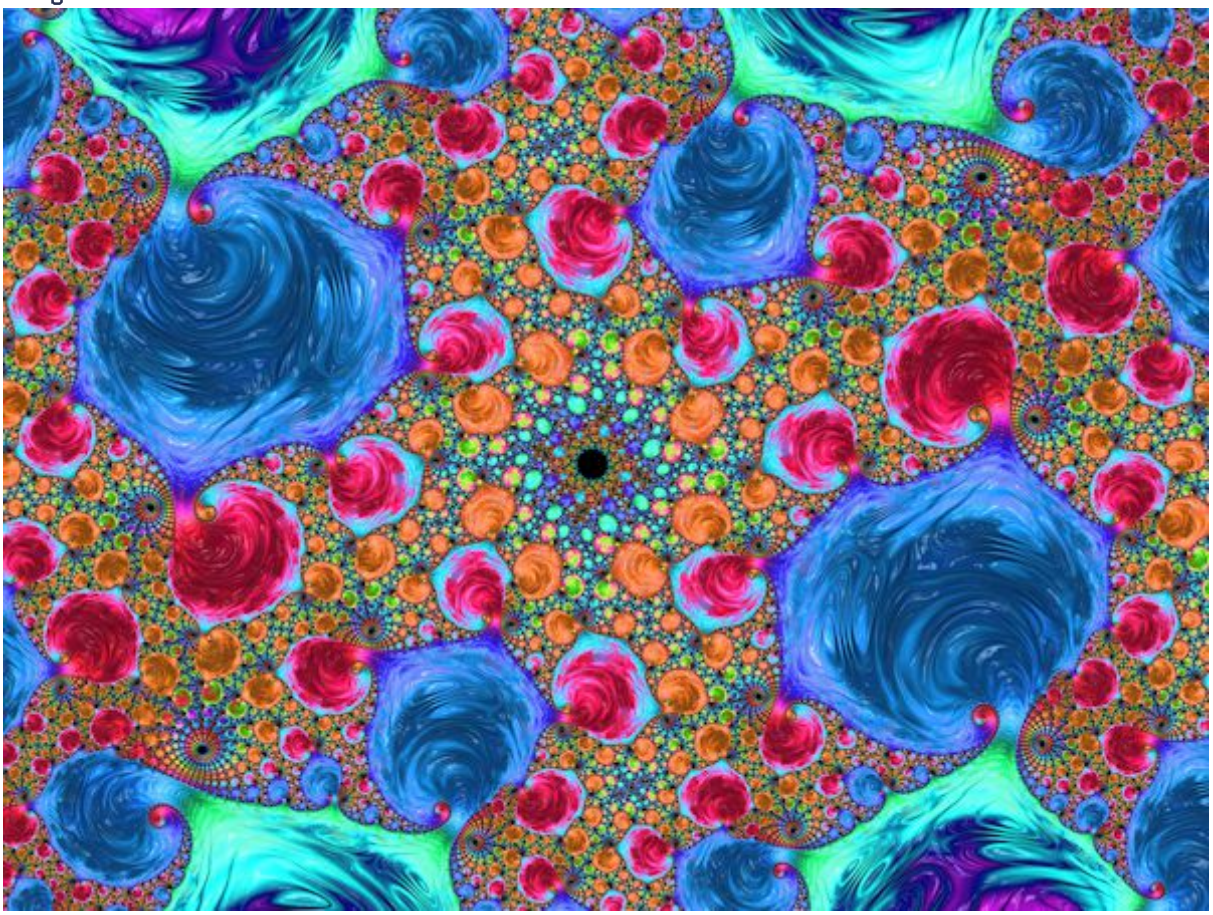
Original6 =

Reconstructed6 =

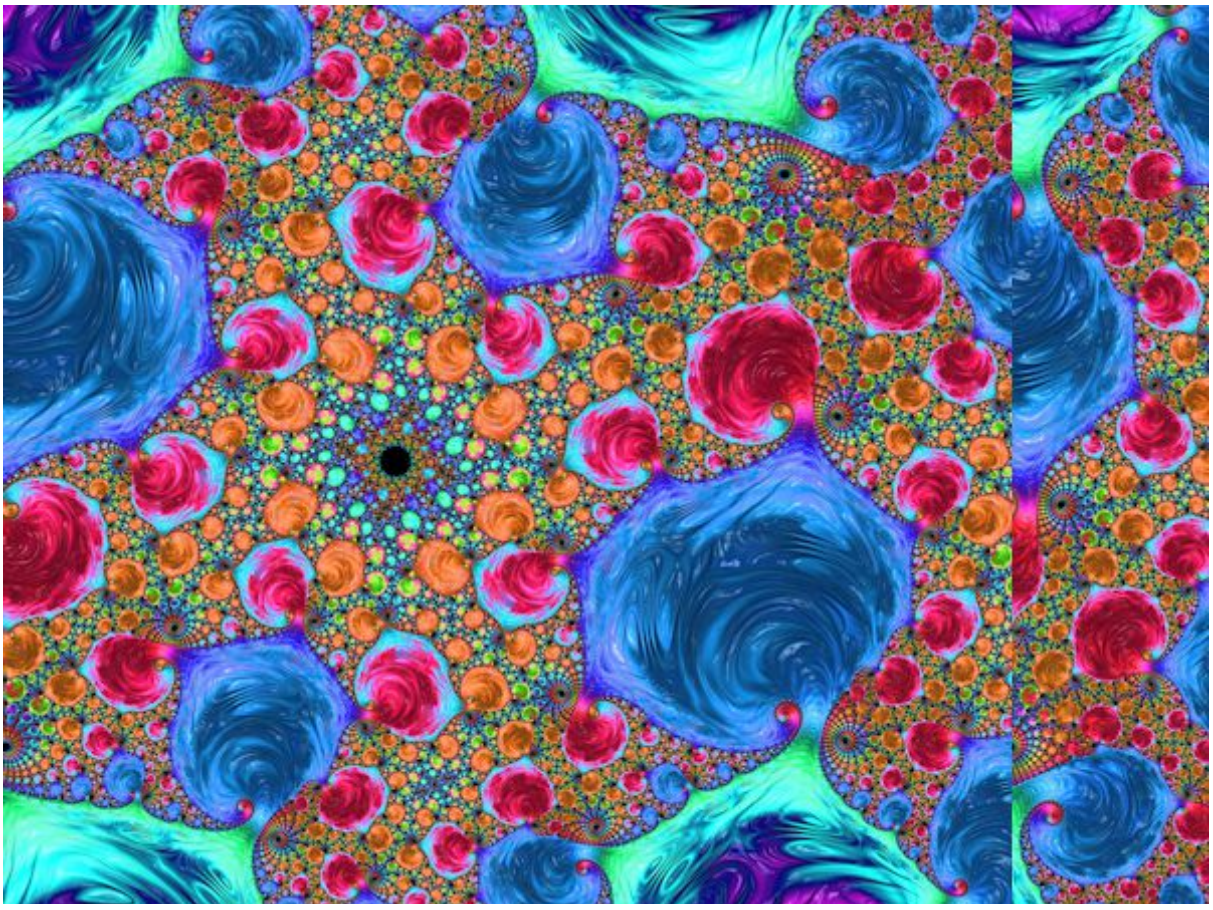
Original7 =

Reconstructed7 =

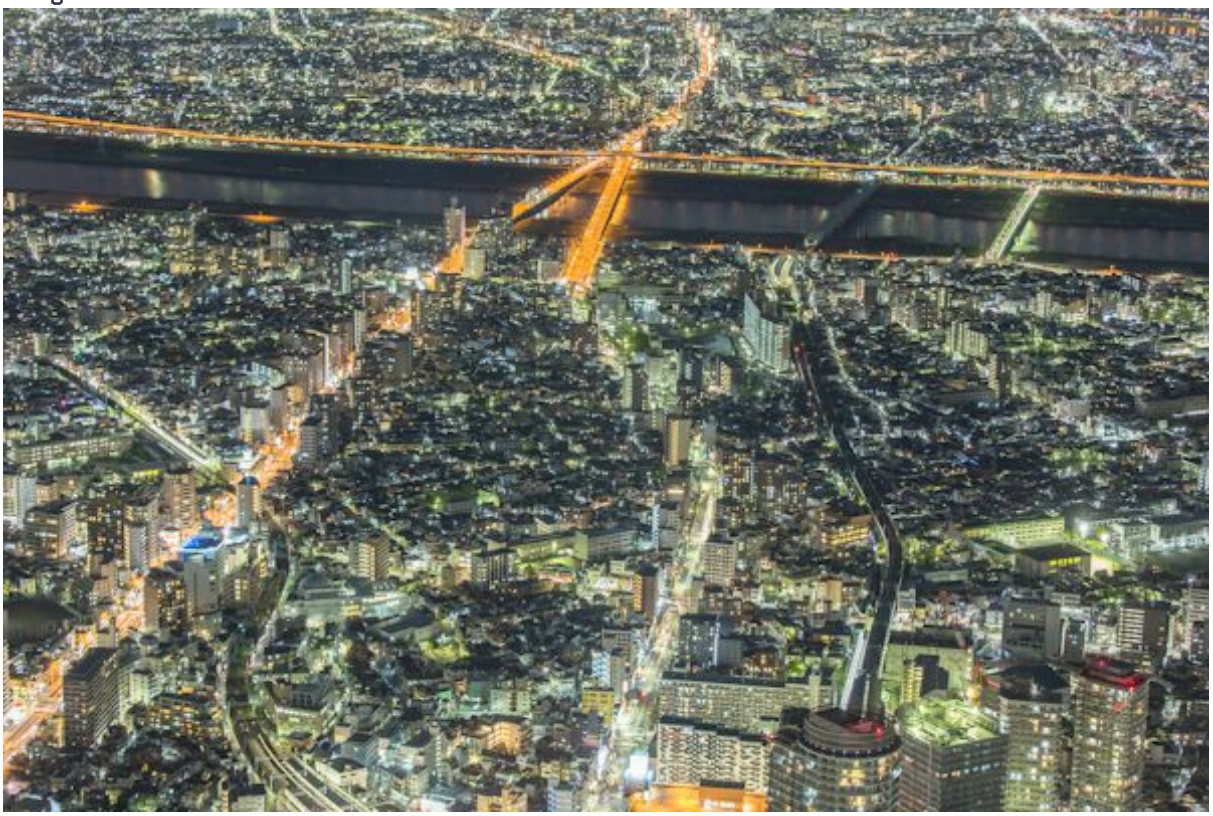
Original8 =



Reconstructed8 =



Original9 =



Reconstructed9 =