```
## s194624.jl

using Random
include("IO.jl")
include("TABU.jl")

struct ArgumentException <: Exception
    message::String
end

function main()
    localSearchTime = 60
    instanceLocation = ARGS[1]
    solutionLocation = ARGS[2]
    totalTime = parse(Int, ARGS[3])

    name, UB, dim, dist = read_instance(instanceLocation)

    println("Dimension: ", dim)
    k = Int(round(dim/4))

    if (k == 0)
        k = 1
    end
    if (length(ARGS) > 3)
        k = parse(Int, ARGS[4])
    end
    println("k: ", k)

    diversifyFrequency = 1

    println("Running instance: ", name)
    println(string("Upper bound: ", UB))

    if (ARGS[2] == " ")
        vals = rsplit(name, ".", limit=2)
        solutionLocation = string("sols/", vals[1], ".sol")
    end

    # Initialize with solution using nearest neighbor
    println("Finding initial solution...")
    s, objectiveValue = nearestNeighbor(dist, dim)
    println("Initial solution found")

    # Find the initial local minimum
    iterations = 0
    elapsedTime = 0

    # Perform iterated local search
    println("Allowed time: ", totalTime, " seconds")

    bestSolution = copy(s)
    bestObjectiveValue = objectiveValue

    previousMove = (-1,-1)
    visitedSolutions = [s]

    noLegalNeighbors = true

    updates = 1
    shuffleFrequency = 10
    switch = 0

    lastUpdateTime = elapsedTime
    start = time_ns()
    while (elapsedTime < totalTime)

        s, objectiveValue, noLegalNeighbors = BestNonTABU(s, objectiveValue, dim, dist, visitedSolutions)
        visitSolution(visitedSolutions, s, k)
        if (objectiveValue < bestObjectiveValue)
            bestSolution = copy(s)
            bestObjectiveValue = objectiveValue
            println()
            println("Time: ", elapsedTime, " seconds")
            println("New best: ", bestObjectiveValue)
```

```
            println("Upper bound: ", UB)
            lastUpdateTime = elapsedTime
        end

        if (noLegalNeighbors || (elapsedTime - lastUpdateTime) > diversifyFrequency)
            # Diversification
            swaps = Int(ceil(dim/2))
            for i in 1:swaps
                edgeA, edgeB = getRandomEdgePair(dim)
                s, objectiveValue = twoOpt(s, objectiveValue, edgeA, edgeB, dist)
            end

            if (updates % shuffleFrequency == 0)
                shuffle!(s)
            end

            updates += 1
            s = makeFeasible(s, dist)
            objectiveValue = getObjectiveValue(s, dist)
            lastUpdateTime = elapsedTime
        end

        iterations += 1
        elapsedTime = round((time_ns()-start)/1e9,digits=3)
    end

    println("\nSearch completed.")
    println(string(iterations, " total iterations"))
    println(string("Final objective value: ", bestObjectiveValue))
    println(string("Upper bound: ", UB))

    writeSolution(bestSolution, solutionLocation)
end
main()
```

```julia
## IO.jl

# Read an instance of the Clever Traveling Salesperson Problem
# Input: filename = path + filename of the instance
function read_instance(filename)
    # Opens the file
    f = open(filename)
    # Reads the name of the instance
    name = split(readline(f))[2]
    # reads the upper bound value
    upper_bound = parse(Int64,split(readline(f))[2])
    readline(f)#type
    readline(f)#comment
    # reads the dimentions of the problem
    dimention = parse(Int64,split(readline(f))[2]);
    readline(f)#Edge1
    readline(f)#Edge2
    readline(f)#Edge3
    readline(f)#Dimention 2

    # Initialises the cost matrix
    cost = zeros(Int64,dimention,dimention)
    # Reads the cost matrix
    for i in 1:dimention
        data = parse.(Int64,split(readline(f)))
        cost[i,:]=data
    end

    # Closes the file
    close(f)

    # Returns the input data
    return name, upper_bound, dimention, cost
end

function writeSolution(solution, solutionLocation)
    wDir = string(pwd())

    dir, file = splitdir(solutionLocation)
    if (!isdir(dir))
        mkpath(string("./", dir, "/"))
    end

    open(string(wDir, "/", solutionLocation), "w") do f
        for i in eachindex(solution)
            write(f, string(solution[i]-1, " "))
        end
    end
end
```

```
## TABU.jl

function insertElement(list, element, dist)
    n = size(list)[1]
    for i in n:-1:1
        if (dist[element,list[i]] == -1)
            insert!(list, i+1, element)
            return list
        end
    end
    insert!(list, 1, element)
    return list
end

function makeFeasible(s, dist)
    n = length(s)
    solution = [s[n]]
    for i in n-1:-1:1
        solution = insertElement(solution, s[i], dist)
    end
    return solution
end

function checkNeighbors(ID, visited, dist, dim)
    max = maximum(dist)+1
    currentList = zeros(dim)

    for i in eachindex(currentList)
        if i in visited
            currentList[i] = max
        else
            currentList[i] = dist[ID,i]
        end
    end
    return findmin(currentList)
end

function nearestNeighbor(dist, dim)
    visited = zeros(Int,1)
    visited[1] = 1

    nextNode = 1
    while(size(visited)[1] < dim)
        shortDist, nextNode = checkNeighbors(nextNode, visited, dist, dim)
        append!(visited,nextNode)
    end
    solution = makeFeasible(visited, dist)
    return solution, getObjectiveValue(solution, dist)
end

# Calculates the objective value of a given solution iteratively
function getObjectiveValue(solution, dist)
    val = 0
    n = size(solution)[1]
    for i in 1:n-1
        val = val + dist[solution[i], solution[i+1]]
    end
    return val
end

function legalPair(n, m)
    if (abs(n-m) < 2)
        return false
    end

    return true
end

function twoOpt(s, objectiveValue, edgeA, edgeB, dist)
    newSolution = copy(s)
    newSolution[edgeA+1], newSolution[edgeB] = newSolution[edgeB], newSolution[edgeA+1]
    newObjectiveValue = swapObjectiveValue(s, objectiveValue, edgeA, edgeB, dist)
    return newSolution, newObjectiveValue
end
```

```
# Given two edges 'n' and 'm' recalculate the relevant costs for the new solution
function swapObjectiveValue(s, sum, n, m, dist)
    s = copy(s)
    if (abs(n-m) == 2)
        sum = sum - (dist[s[n],s[n+1]] + dist[s[n+1],s[m]]
                   + dist[s[m],s[m+1]])
        sum = sum + (dist[s[n],s[m]] + dist[s[m],s[n+1]]
                   + dist[s[n+1],s[m+1]])
    else
        sum = sum - (dist[s[n],s[n+1]] + dist[s[n+1],s[n+2]]
                   + dist[s[m-1],s[m]] + dist[s[m],s[m+1]])
        sum = sum + (dist[s[n],s[m]] + dist[s[m],s[n+2]]
                   + dist[s[m-1],s[n+1]] + dist[s[n+1],s[m+1]])
    end
    return sum
end

# Returns true if solutions are equal, otherwise false
function compareSolutions(sol1, sol2)
    for i in eachindex(sol1)
            if (sol1[i] != sol2[i])
                return false
            end
        end
    return true
end

function solutionVisited(s, visitedSolutions)
    for prevSolution in visitedSolutions
        if (compareSolutions(s, prevSolution))
            return true
        end
    end
    return false
end

function isLegal(s, dist)
    for i in eachindex(s)
        for j in i:length(s)
            if (dist[s[i],s[j]] == -1)
                return false
            end
        end
    end
    return true
end

function beenVisited(OGSolution, previousMoves, dist)
    s = copy(OGSolution)
    for move in previousMoves
        newSolution, newObjectiveValue = twoOpt(s, objectiveValue, move[1], move[2], dist)
    end
end

function BestNonTABU(originalSolution, originalObjectiveValue, dim, dist, visitedSolutions)
    s = copy(originalSolution)
    objectiveValue = copy(originalObjectiveValue)
    noLegalNeighbors = true
    for i in 1:dim-1
        for j in i:dim-1
            if(legalPair(i, j))
                newSolution, newObjectiveValue = twoOpt(originalSolution, originalObjectiveValue, i, j, dist)
                if (isLegal(newSolution, dist) && !solutionVisited(newSolution, visitedSolutions))
                    if (noLegalNeighbors || newObjectiveValue < objectiveValue)
                        s = newSolution
                        objectiveValue = newObjectiveValue
                    end
                    noLegalNeighbors = false
                end
            end
        end
    end
    return s, objectiveValue, noLegalNeighbors
end
```

```
function getRandomEdgePair(dim)
    population = [i for i in 1:dim-1]
    edgeA = population[rand(1:length(population))]
    filter!(x -> abs(edgeA - x) > 2, population)
    edgeB = population[rand(1:length(population))]
    if (edgeB < edgeA)
        edgeA, edgeB = edgeB, edgeA
    end
    return edgeA, edgeB
end

function visitSolution(visitedSolutions, s, k)
    n = length(visitedSolutions)
    if (n == k)
        deleteat!(visitedSolutions, 1)
    end
    push!(visitedSolutions, s)
end
```