Julia is a language meant for working with data. In that regard, we would like to be able to be able to perform analyses like linear regression and hypothesis testing. We would also like to perform more complex operations including building neural networks and running algorithms. Thankfully, there are packages that allow us to easily perform these tasks.

In this presentation, I will be focusing on three different packages; MLJ.jl, Graphs.jl, and JuMP.jl.

MLJ.jl is a high-level machine learning framework, while Graphs.jl is meant for network analysis. JuMP.jl is a mathematical optimization framework.

MLJ is a machine learning package that is similar to Python's scikit-learn, due to its breadth and flexibility in performing different kinds of analyses.

```
using Pkg
Pkg.add(["MLJ", "MLJLinearModels", "DataFrames", "StatsBase"])
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

There are numerous models we can configure and implement for many different machine learning problems, as shown by listing out the possible models.

```
using MLJ
models()
```

```
234-element Vector{NamedTuple{(:name, :package_name, :is_supervised, :abstract_type, :constru
 (name = ABODDetector, package_name = OutlierDetectionNeighbors, ... )
 (name = ABODDetector, package_name = OutlierDetectionPython, ... )
 (name = ARDRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostClassifier, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = AdaBoostStumpClassifier, package_name = DecisionTree, ... )
 (name = AffinityPropagation, package_name = Clustering, ... )
 (name = AffinityPropagation, package_name = MLJScikitLearnInterface, ... )
 (name = AgglomerativeClustering, package_name = MLJScikitLearnInterface, ... )
 (name = AutoEncoder, package_name = BetaML, ... )
 (name = BM25Transformer, package_name = MLJText, ... )
 (name = BaggingClassifier, package_name = MLJScikitLearnInterface, ... )
 (name = BaggingRegressor, package_name = MLJScikitLearnInterface, ... )

 (name = TSVDTransformer, package_name = TSVD, ... )
```

```
 (name = TfidfTransformer, package_name = MLJText, ... )
 (name = TheilSenRegressor, package_name = MLJScikitLearnInterface, ... )
 (name = TomekUndersampler, package_name = Imbalance, ... )
 (name = UnivariateBoxCoxTransformer, package_name = MLJModels, ... )
 (name = UnivariateDiscretizer, package_name = MLJModels, ... )
 (name = UnivariateFillImputer, package_name = MLJModels, ... )
 (name = UnivariateStandardizer, package_name = MLJModels, ... )
 (name = UnivariateTimeTypeToContinuous, package_name = MLJModels, ... )
 (name = XGBoostClassifier, package_name = XGBoost, ... )
 (name = XGBoostCount, package_name = XGBoost, ... )
 (name = XGBoostRegressor, package_name = XGBoost, ... )
```

One example that we can look at is implementing linear regression on the Boston housing dataset. We would like to model the prices of properties as functions of the other variables.

We start by loading the dataset and then dividing the data into training and testing data. This can be done without any external tools.

```
using MLJ, DataFrames, MLJLinearModels, StatsBase

# Load dataset
X, y = @load_boston

# Convert X to a DataFrame
X = DataFrame(X)

# Split data into training (80%) and test (20%) sets
train, test = partition(eachindex(y), 0.8, shuffle=true)
X_train, X_test = X[train, :], X[test, :]
y_train, y_test = y[train], y[test]
```

```
([23.4, 21.2, 5.0, 18.9, 16.1, 19.4, 28.1, 19.7, 6.3, 9.5  …  17.8, 17.5, 11.9, 36.0, 29.6, 5
```

Now, we load an MLJ model. Specifically, we will use LinearRegressor, which is a simple linear regression model. MLJ offers other alternatives, including RidgeRegressor and LassoRegressor, as well as linear model classifiers like LogisticClassifier.

```
# Load Linear Regression model
@load LinearRegressor pkg=MLJLinearModels
model = LinearRegressor()

# Create a machine (model + data)
```

```
mach = machine(model, X_train, y_train)

# Train the model
fit!(mach)
```

```
[ Info: For silent loading, specify `verbosity=0`.

import MLJLinearModels

[ Info: Training machine(LinearRegressor(fit_intercept = true, …), …).
 Info: Solver: Analytical
   iterative: Bool false
   max_inner: Int64 200

trained Machine; caches model-specific representations of data
  model: LinearRegressor(fit_intercept = true, …)
  args:
    1:  Source @471   Table{AbstractVector{Continuous}}
    2:  Source @640   AbstractVector{Continuous}
```

After fitting the model, we can now make predictions to see how accurate our model is! We can use the StatsBase package to calculate metrics like R-squared and MAE.

```
# Make predictions
y_pred = MLJ.predict(mach, X_test)

r2_score = cor(y_pred_numeric, y_test)^2
mae_score = mean(abs.(y_pred_numeric - y_test))

println("R² Score: ", r2_score)
println("MAE: ", mae_score)
```

```
R² Score: 0.7778171789206735
MAE: 3.5654545702415246
```

On the other hand, if you are interested in solving problems with deep learning techniques, Flux.jl is a popular option for that. Flux is a deep learning framework that has a lot of the same functionalities as MLJ, but is more focused on deep learning and allowing the user to control those areas of computation.

MLJ is more suited towards high-level implementations. It supports diverse approaches to problems, and allows users to pull together and integrate multiple models, without requiring low-level mastery. However, it does not inherently support GPU acceleration, which makes it slower when running large tasks.

The last two packages I want to go over are Graphs.jl and JuMP.jl. While MLJ and Flux are meant for machine learning, there may be times when a user is interested in implementing optimization algorithms, like Dijkstra's or Simplex. These algorithms are of a different flavor as they are not necessarily meant to solve prediction or inference problems, but are many times useful for finding optimal paths or augmentations for a specific task.

We can see how Graphs.jl can be useful through an implementation of Dijkstra's Algorithm.

Below is a raw implementation of the algorithm on a simple graph, with no external packages being used.

```
function dijkstra(graph::Dict{Int, Dict{Int, Int}}, start::Int)
    # Initialize distances to infinity, except for the start node
    dist = Dict(k => Inf for k in keys(graph))
    dist[start] = 0

    # Track visited nodes
    visited = Dict(k => false for k in keys(graph))

    # Track shortest paths
    prev = Dict{Int, Union{Nothing, Int}}(k => nothing for k in keys(graph))

    while true
        # Select the unvisited node with the smallest distance
        u = nothing
        for node in keys(graph)
            if !visited[node] && (u === nothing || dist[node] < dist[u])
                u = node
            end
        end

        # Stop if there are no reachable unvisited nodes
        if u === nothing || dist[u] == Inf
            break
        end

        # Mark node as visited
        visited[u] = true
```

```
        # Update distances to neighbors
        for (v, weight) in graph[u]
            alt = dist[u] + weight
            if alt < dist[v]
                dist[v] = alt
                prev[v] = u
            end
        end
    end

    return dist, prev
end

# Function to reconstruct the shortest path
function shortest_path(prev::Dict{Int, Union{Nothing, Int}}, target::Int)
    path = []
    while target !== nothing
        pushfirst!(path, target)
        target = prev[target]
    end
    return path
end

# Define a weighted graph using a properly typed dictionary
graph = Dict{Int, Dict{Int, Int}}(
    1 => Dict(2 => 4, 3 => 1),
    2 => Dict(4 => 1),
    3 => Dict(2 => 2, 4 => 5),
    4 => Dict(5 => 3),
    5 => Dict()
)

# Run Dijkstra's algorithm from node 1
distances, predecessors = dijkstra(graph, 1)

# Print shortest distance from node 1 to 5
println("Shortest distance from 1 to 5: ", distances[5])

# Get the shortest path from node 1 to 5
path = shortest_path(predecessors, 5)
println("Shortest path from 1 to 5: ", path)
```

```
Shortest distance from 1 to 5: 7.0
Any[1, 3, 2, 4, 5] 1 to 5:
```

The code is quite bulky and requires a lot of work to develop. By using the Graphs package, we can implement a shorter, more elegant solution to the same problem.

```
using Pkg
Pkg.add("Graphs")
using Graphs
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```
# Create a directed graph with 5 nodes
g = SimpleDiGraph(5)

# Define weighted edges
edges = [
    (1, 2, 4), (1, 3, 1),
    (3, 2, 2), (3, 4, 5),
    (2, 4, 1), (4, 5, 3)
]

# Add edges to the graph
for (u, v, _) in edges
    add_edge!(g, u, v)  # Add edge
end

# Create a weight matrix (initialize with Inf)
weights = fill(Inf, nv(g), nv(g))

# Assign weights to the adjacency matrix
for (u, v, w) in edges
    weights[u, v] = w
end

# Run Dijkstra's Algorithm from node 1
result = dijkstra_shortest_paths(g, 1, weights)

# Get the shortest path distance to node 5
```

```julia
println("Shortest distance from node 1 to 5: ", result.dists[5])

# Retrieve the shortest path to node 5
function get_path(result, target)
    path = []
    while target != 0
        pushfirst!(path, target)
        target = result.parents[target]
    end
    return path
end

best_path = get_path(result, 5)
println("Shortest path from node 1 to 5: ", best_path)
```

```
Shortest distance from node 1 to 5: 7.0
Shortest path from node 1 to 5: Any[1, 3, 2, 4, 5]
```

We can see that the Graphs package is quite useful for the representation of non-numeric objects. Networks are the foundation of many important problems, and combining them with the numerical computing power of Julia can help to solve many problems.

Finally, we will go over a mathematical optimization framework called JuMP.jl, as well as a mathematical solver package called Gurobi. Mathematical optimization is at the core of many different problems like portfolio optimization, control theory, and other various fields. Being able to solve optimization problems as fast as possible carries a lot of importance in fields like supply chain logistics, asset allocation, etc.

JuMP works within Julia but acts like its own language. By defining objects like objective functions and constraints, JuMP becomes a versatile tool for constructing and solving problems like Linear Programs, Quadratic Programs, and Mixed Integer Optimization problems.

On the flip side, solvers like Gurobi are the workhorses for solving these problems. Gurobi in particular is the world's fastest mathematical solver, due to its ability to adjust its solving strategies to each problem as well as the extreme level of care put into optimizing each process through parallel computing and fast code.

Gurobi is also free for academics!

```julia
Pkg.add("JuMP")
Pkg.add("Gurobi")
Pkg.add("MathOptInterface")
```

```
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Project.toml`
  No Changes to `C:\Users\joshu\.julia\environments\v1.11\Manifest.toml`
```

```
using JuMP, Gurobi, MathOptInterface
env = Gurobi.Env()
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
```

```
Gurobi.Env(Ptr{Nothing} @0x000001b379515120, false, 0)
```

```
# Define a JuMP model with Gurobi as the solver
model = JuMP.Model(Gurobi.Optimizer)
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
```

```
A JuMP Model
  solver: Gurobi
  objective_sense: FEASIBILITY_SENSE
  num_variables: 0
  num_constraints: 0
  Names registered in the model: none
```

Let's say that we have a problem of the form:

$$\max_{x_1, x_2} 5x_1 + 4x_2$$

With the constraints:

$$6x_1 + 4x_2 \leq 24$$
$$x_1 + 2x_2 \leq 6$$
$$x_1, x_2 \geq 0$$

We can define this problem with JuMP and subsequently solve!

```
# Define variables (x1, x2 >= 0)
@variable(model, x1 >= 0)
@variable(model, x2 >= 0)

# Objective function: Maximize 5x1 + 4x2
@objective(model, Max, 5x1 + 4x2)

# Constraints
@constraint(model, 6x1 + 4x2 <= 24)
@constraint(model, x1 + 2x2 <= 6)

# Solve the optimization problem
optimize!(model)

# Display results
println("Optimal x1 = ", value(x1))
println("Optimal x2 = ", value(x2))
println("Optimal objective value = ", objective_value(model))
```

```
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0x9f631cdb
Coefficient statistics:
  Matrix range     [1e+00, 6e+00]
  Objective range  [4e+00, 5e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 2e+01]
Presolve time: 0.01s
Presolved: 2 rows, 2 columns, 4 nonzeros
```

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    9.0000000e+30   2.750000e+30   9.000000e+00      0s
       2    2.1000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.02 seconds (0.00 work units)
Optimal objective  2.100000000e+01

User-callback calls 47, time in user-callback 0.00 sec
Optimal x1 = 3.0
Optimal x2 = 1.5
Optimal objective value = 21.0
```

JuMP can also be used to solve other genres of optimization problems. For example, let us consider an MIO problem of the form:

$$\max_{x_1, x_2} 2x_1 + 3x_2$$

$$4x_1 + 3x_2 \leq 12$$

$$2x_1 + x_2 \leq 6$$

$$x_1, x_2 \geq 0$$

$$x_1, x_2 \in \mathbb{Z}$$

```julia
using JuMP, Gurobi

model = JuMP.Model(Gurobi.Optimizer)

@variable(model, x1 >= 0, Int) # Must specify as Int
@variable(model, x2 >= 0, Int) # Must specify as Int

@objective(model, Max, 2x1 + 3x2)

@constraint(model, 4x1 + 3x2 <= 12)
@constraint(model, 2x1 + x2 <= 6)

optimize!(model)
```

```julia
println("Optimal x1 = ", value(x1))
println("Optimal x2 = ", value(x2))
println("Optimal objective value = ", objective_value(model))
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xb17432a7
Variable types: 0 continuous, 2 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 4e+00]
  Objective range  [2e+00, 3e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 1e+01]
Found heuristic solution: objective 6.0000000
Presolve removed 2 rows and 2 columns
Presolve time: 0.02s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.03 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 2: 12 6

Optimal solution found (tolerance 1.00e-04)
Best objective 1.200000000000e+01, best bound 1.200000000000e+01, gap 0.0000%

User-callback calls 198, time in user-callback 0.00 sec
-0.0mal x1 =
Optimal x2 = 4.0
Optimal objective value = 12.0
```

We can try a large problem. MIO is an especially difficult problem to solve, and when we have a lot of variables and constraints to fulfill, it can take more and more time to solve. JuMP and Gurobi offer functionalities useful in working around potentially nasty problems; we can

adjust the number of the threads being used, and we can tell the solver that we are willing to accept certain levels of optimality gaps, in order to speed up solving.

```
using JuMP, Gurobi, Random

# Use MOI directly from MathOptInterface (no need to redefine it)
Random.seed!(42)

# Create the model using Gurobi
model = JuMP.Model(Gurobi.Optimizer)

# Problem size parameters
num_int_vars = 300  # Integer variables
num_bin_vars = 150  # Binary variables
num_constraints = 500  # Total constraints

# Generate random coefficients ensuring feasibility & boundedness
c = rand(10:100, num_int_vars)      # Objective coefficients for x
d = rand(5:50, num_bin_vars)        # Objective coefficients for y
e = rand(1:5, num_int_vars)         # Small penalty to prevent unbounded solutions
a = rand(1:5, num_constraints, num_int_vars)  # Constraint coefficients for x (smaller values
b = rand(1:3, num_constraints, num_bin_vars)  # Constraint coefficients for y (smaller values
k = sum(a, dims=2)[:, 1] * 50 + sum(b, dims=2)[:, 1] * 1  # Ensures feasibility

# Define integer variables (1   x_i   100, x_i   )
@variable(model, 1 <= x[1:num_int_vars] <= 100, Int)

# Define binary variables (y_j   {0,1})
@variable(model, y[1:num_bin_vars], Bin)

# Define the objective function (maximize Z)
@objective(model, Max,
    sum(c[i] * x[i] for i in 1:num_int_vars) +
    sum(d[j] * y[j] for j in 1:num_bin_vars) -
    sum(e[i] * x[i]^2 for i in 1:num_int_vars)  # Small quadratic penalty to ensure boundedne
)

# Add constraints ensuring feasibility
for j in 1:num_constraints
    @constraint(model,
        sum(a[j, i] * x[i] for i in 1:num_int_vars) +
        sum(b[j, j] * y[j] for j in 1:num_bin_vars) <= k[j]
    )
```

```
end

# Increase solver difficulty while keeping it feasible
set_optimizer_attribute(model, "TimeLimit", 120)  # Give Gurobi 2 minutes
set_optimizer_attribute(model, "MIPGap", 0.02)  # Allow 2% optimality gap
# set_optimizer_attribute(model, "Threads", 1)  # Single-threaded (slows down solving)
# set_optimizer_attribute(model, "Presolve", 2)  # Enable presolve for better performance
# set_optimizer_attribute(model, "Cuts", 1)  # Allow cuts to help find solutions

# Solve the problem and track time
@time optimize!(model)

println("Optimal Objective Value: ", objective_value(model))
println("First 10 integer variable values: ", [value(x[i]) for i in 1:10])
println("First 10 binary variable values: ", [value(y[j]) for j in 1:10])
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
Academic license - for non-commercial use only - expires 2026-03-02
Set parameter TimeLimit to value 120
Set parameter MIPGap to value 0.02
Set parameter MIPGap to value 0.02
Set parameter TimeLimit to value 120
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Non-default parameters:
TimeLimit  120
MIPGap  0.02

Optimize a model with 500 rows, 450 columns and 225000 nonzeros
Model fingerprint: 0xe3ee7d39
Model has 300 quadratic objective terms
Variable types: 0 continuous, 450 integer (150 binary)
Coefficient statistics:
  Matrix range     [1e+00, 5e+00]
  Objective range  [5e+00, 1e+02]
  QObjective range [2e+00, 1e+01]
  Bounds range     [1e+00, 1e+02]
  RHS range        [4e+04, 5e+04]
```

```
Found heuristic solution: objective 15690.000000
Presolve added 1 rows and 0 columns
Presolve removed 0 rows and 58 columns
Presolve time: 0.42s
Presolved: 501 rows, 392 columns, 150592 nonzeros
Presolved model has 300 quadratic objective terms
Variable types: 0 continuous, 392 integer (50 binary)
Found heuristic solution: objective 127175.00000

Root relaxation: objective 1.314432e+05, 95 iterations, 0.03 seconds (0.01 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 131443.208    0  229 127175.000 131361.000  3.29%     -    0s
H    0     0                    131192.00000 131361.000  0.13%     -    0s

Explored 1 nodes (95 simplex iterations) in 0.60 seconds (0.16 work units)
Thread count was 8 (of 8 available processors)

Solution count 3: 131192 127175 15690

Optimal solution found (tolerance 2.00e-02)
Best objective 1.311920000000e+05, best bound 1.313610000000e+05, gap 0.1288%

User-callback calls 248, time in user-callback 0.00 sec
  0.862398 seconds (67.22 k allocations: 13.035 MiB, 25.20% compilation time)
Optimal Objective Value: 131192.0
[12.0, 13.0, 14.0, 13.0, 36.0, 7.0, 33.0, 7.0, 26.0, 5.0]
First 10 binary variable values: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Finally, let's look at an application of JuMP to a more realistic problem. We will look at the Traveling Salesman Problem, where we want to find the shortest route that visits all nodes . This problem is notoriously difficult to solve because there are so many candidate solutions.

```julia
using JuMP, Gurobi, Plots, Random

# Function to generate random city coordinates
function generate_cities(n, seed=123)
    Random.seed!(seed)
    return [(rand(), rand()) for _ in 1:n]
end
```

```
# Compute Euclidean distance matrix
function compute_distance_matrix(cities)
    n = length(cities)
    dist_matrix = zeros(n, n)
    for i in 1:n, j in 1:n
        dist_matrix[i, j] = hypot(cities[i][1] - cities[j][1], cities[i][2] - cities[j][2])
    end
    return dist_matrix
end

# Solve TSP using JuMP and Gurobi
function solve_tsp(dist_matrix)
    n = size(dist_matrix, 1)
    model = JuMP.Model(Gurobi.Optimizer)

    @variable(model, x[1:n, 1:n], Bin)

    # Objective: Minimize travel distance
    @objective(model, Min, sum(dist_matrix[i, j] * x[i, j] for i in 1:n, j in 1:n))

    # Constraints: Each city must be entered and exited exactly once
    @constraint(model, [i in 1:n], sum(x[i, j] for j in 1:n if i != j) == 1)
    @constraint(model, [j in 1:n], sum(x[i, j] for i in 1:n if i != j) == 1)

    # Subtour elimination (MTZ formulation)
    @variable(model, u[2:n] >= 0)
    @constraint(model, [i in 2:n, j in 2:n; i != j],
                u[i] - u[j] + n * x[i, j]  n - 1)

    optimize!(model)

    if termination_status(model) == MOI.OPTIMAL
        println("Optimal tour found with cost: ", objective_value(model))
        tour = Dict{Int, Int}()
        for i in 1:n, j in 1:n
            if value(x[i, j]) > 0.5
                tour[i] = j
            end
        end
        return tour
    else
        println("No optimal solution found.")
```

```
        return nothing
    end
end

# Function to extract the ordered tour from the dictionary
function get_tour_sequence(tour)
    if tour === nothing
        return []
    end
    n = length(tour)
    sequence = [1]  # Start from node 1
    while length(sequence) < n
        sequence = push!(sequence, tour[sequence[end]])
    end
    push!(sequence, 1)  # Return to the starting city
    return sequence
end

# Function to plot the TSP solution
function plot_tsp(cities, tour_sequence)
    x_vals = [cities[i][1] for i in tour_sequence]
    y_vals = [cities[i][2] for i in tour_sequence]

    scatter([c[1] for c in cities], [c[2] for c in cities], label="Cities", markersize=5)
    plot!(x_vals, y_vals, arrow=true, label="Tour", linewidth=2, color=:blue)
    title!("Optimal TSP Tour")
end

# Generate cities and solve TSP
n = 100  # Number of cities
cities = generate_cities(n)
dist_matrix = compute_distance_matrix(cities)
tour = solve_tsp(dist_matrix)

# Plot the solution if found
if tour !== nothing
    tour_sequence = get_tour_sequence(tour)
    plot_tsp(cities, tour_sequence)
end
```

```
Set parameter Username
Set parameter LicenseID to value 2630108
```

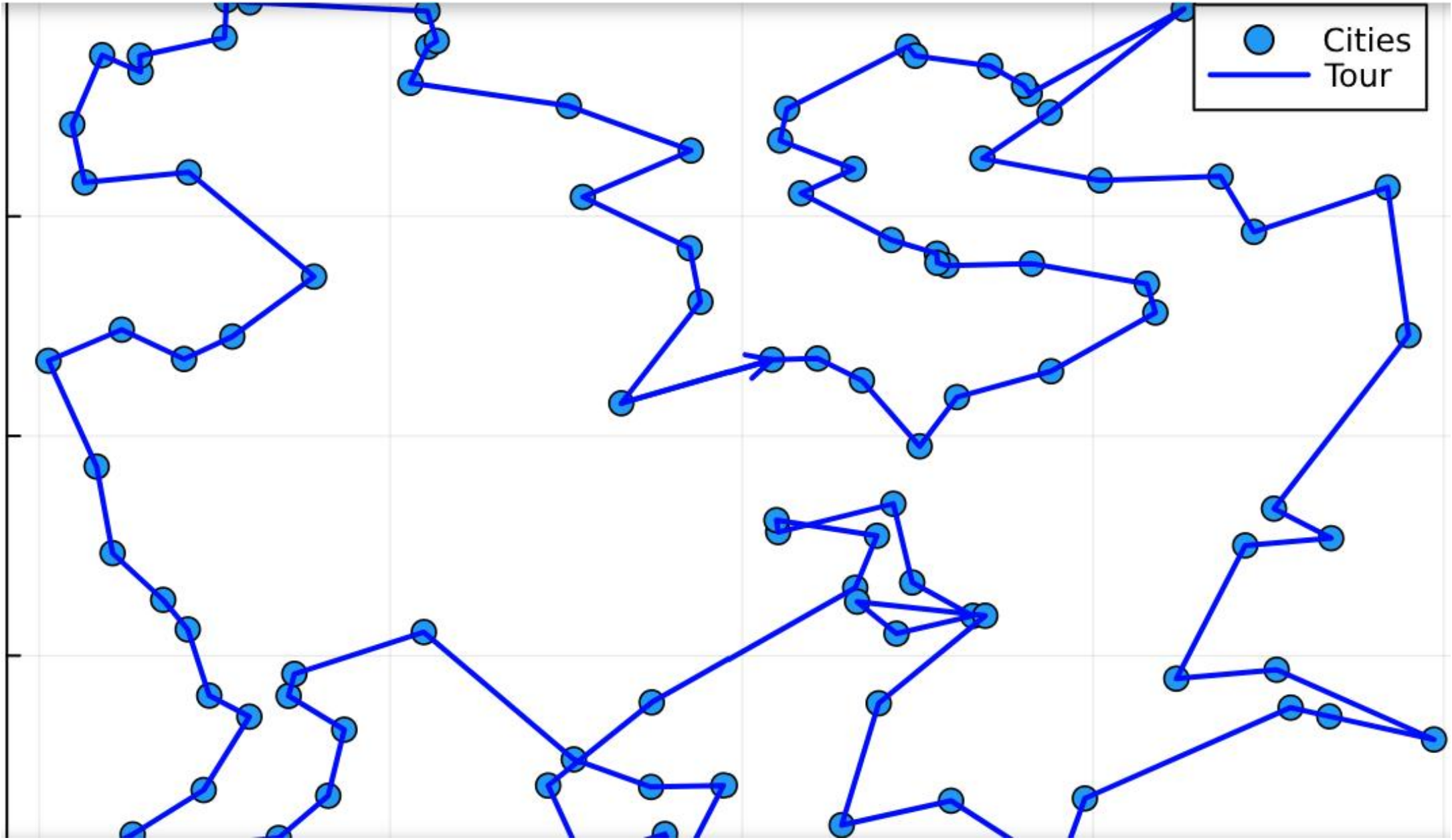
```
Academic license - for non-commercial use only - expires 2026-03-02
Gurobi Optimizer version 12.0.1 build v12.0.1rc0 (win64 - Windows 11.0 (26100.2))

CPU model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, instruction set [SSE2|AVX|AVX2|AV
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 9902 rows, 10099 columns and 48906 nonzeros
Model fingerprint: 0x325090e3
Variable types: 99 continuous, 10000 integer (10000 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+02]
  Objective range  [7e-03, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+02]
Presolve removed 0 rows and 100 columns
Presolve time: 0.10s
Presolved: 9902 rows, 9999 columns, 48906 nonzeros
Variable types: 99 continuous, 9900 integer (9900 binary)

Root relaxation: objective 6.147645e+00, 328 iterations, 0.04 seconds (0.01 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0    6.14765    0  164          -    6.14765      -     -    0s
     0     0    6.84839    0  206          -    6.84839      -     -    0s
     0     0    6.84840    0  202          -    6.84840      -     -    1s
     0     0    6.95682    0  208          -    6.95682      -     -    1s
     0     0    7.00323    0  195          -    7.00323      -     -    1s
     0     0    7.00620    0  196          -    7.00620      -     -    1s
     0     0    7.00622    0  201          -    7.00622      -     -    1s
     0     0    7.03573    0  197          -    7.03573      -     -    2s
     0     0    7.04685    0  198          -    7.04685      -     -    2s
     0     0    7.06155    0  198          -    7.06155      -     -    2s
     0     0    7.14312    0  204          -    7.14312      -     -    2s
     0     0    7.14456    0  204          -    7.14456      -     -    2s
     0     0    7.14456    0  182          -    7.14456      -     -    3s
H    0     0                       14.1468348    7.14595  49.5%     -    3s
     0     0    7.14595    0  182   14.14683    7.14595  49.5%     -    3s
     0     0    7.14595    0  182   14.14683    7.14595  49.5%     -    3s
H    0     0                       11.2270301    7.14595  36.4%     -    6s
H    0     0                       10.8105846    7.14595  33.9%     -    6s
     0     0    7.14595    0  182   10.81058    7.14595  33.9%     -    6s
```

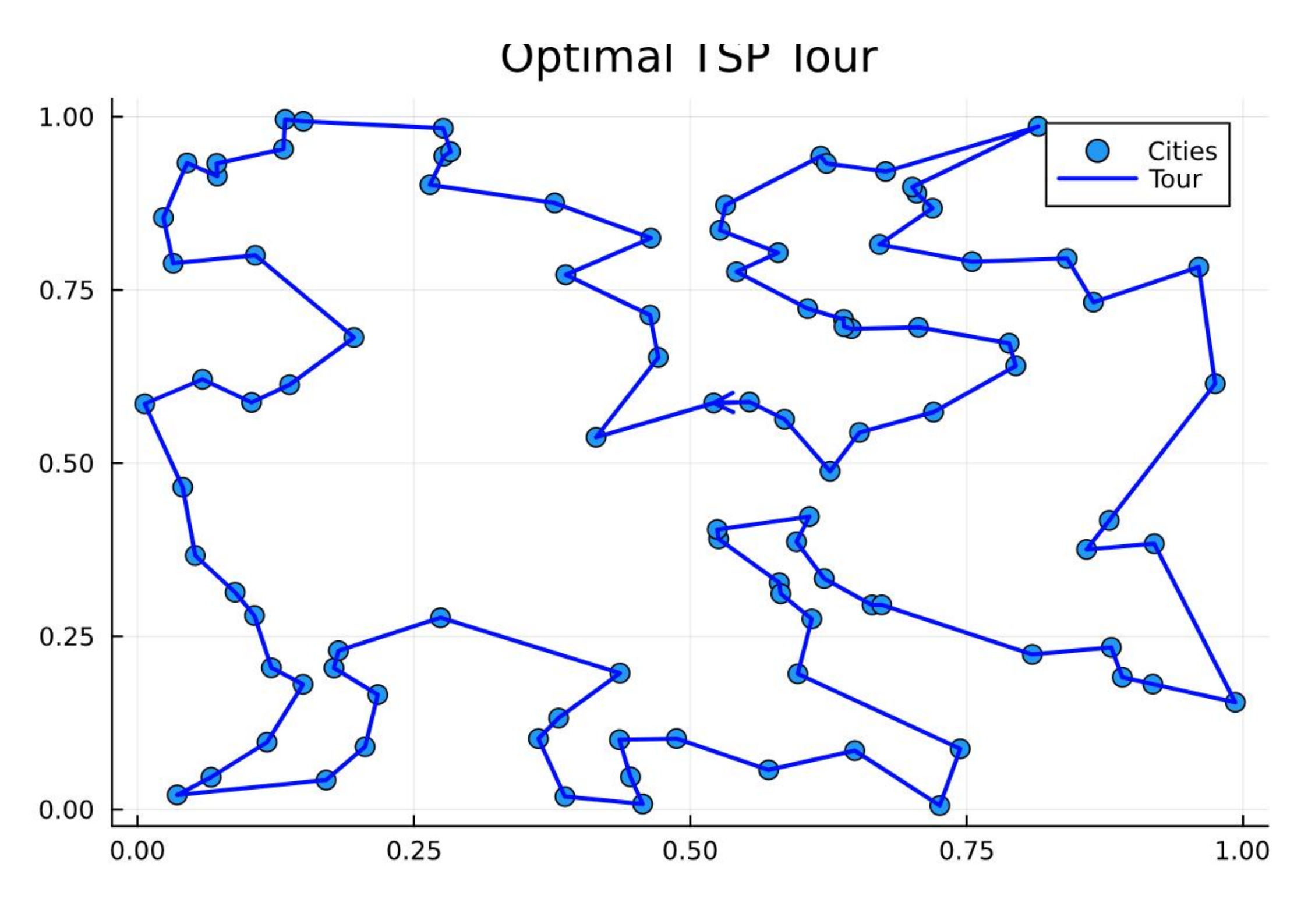
H	0	0				9.8551436	7.15168	27.4%	-	8s
H	0	0				9.8461639	7.15168	27.4%	-	8s
	0	0	7.15168	0	182	9.84616	7.15168	27.4%	-	9s
	0	2	7.15168	0	182	9.84616	7.15168	27.4%	-	9s
	46	52	7.19497	11	138	9.84616	7.15168	27.4%	57.1	10s
H	1775	1720				9.8394427	7.15537	27.3%	12.4	15s
H	2388	2353				9.7590871	7.15537	26.7%	11.7	17s
H	2421	2350				9.7419591	7.15537	26.6%	11.6	17s
H	2543	2450				9.6897273	7.15573	26.2%	11.5	19s
H	2543	2410				9.4987109	7.15573	24.7%	11.5	19s
	2544	2411	7.51069	19	182	9.49871	7.15573	24.7%	11.5	20s
H	2546	2291				9.4753434	7.15573	24.5%	11.4	22s
	2551	2295	7.73303	111	176	9.47534	7.21544	23.9%	11.4	25s
	2561	2301	7.94368	180	169	9.47534	7.26944	23.3%	11.4	30s
	2569	2307	7.42010	123	58	9.47534	7.27258	23.2%	11.3	35s
H	2577	2198				8.1390101	7.27349	10.6%	13.5	40s
H	2580	2090				7.8040188	7.27352	6.80%	13.5	42s
	2587	2096	7.80402	224	173	7.80402	7.27352	6.80%	14.5	45s
H	2587	1991				7.8033637	7.27352	6.79%	14.5	46s
H	2592	1894				7.7907381	7.27352	6.64%	14.5	48s
	2596	1897	7.71989	107	224	7.79074	7.27680	6.60%	14.5	50s
H	2599	1804				7.6702350	7.28059	5.08%	14.5	50s
	2609	1812	7.67023	243	194	7.67023	7.28059	5.08%	16.2	55s
H	2617	1725				7.5608562	7.28493	3.65%	16.1	58s
	2625	1732	7.56086	123	193	7.56086	7.28493	3.65%	17.8	61s
	2636	1742	7.28497	55	31	7.56086	7.28497	3.65%	18.9	65s
H	2673	1681				7.4964226	7.28498	2.82%	20.1	68s
H	2676	1598				7.4897015	7.28498	2.73%	20.2	68s
	2746	1645	7.28843	68	26	7.48970	7.28498	2.73%	21.9	70s
H	2751	1567				7.4858702	7.28498	2.68%	22.1	71s
H	2975	1527				7.4786188	7.28563	2.58%	25.7	74s
H	2976	1456				7.4784553	7.28563	2.58%	25.7	74s
	2980	1458	7.32098	60	217	7.47846	7.28563	2.58%	26.0	75s
	3532	1738	7.32683	72	87	7.47846	7.28696	2.56%	28.3	80s
	4493	2092	7.46343	82	165	7.47846	7.28902	2.53%	31.5	85s
	5258	2273	7.38871	93	108	7.47846	7.29077	2.51%	33.5	90s
	6018	2389	7.34172	74	165	7.47846	7.29351	2.47%	35.3	95s
	6027	2395	7.34470	64	196	7.47846	7.29351	2.47%	35.3	100s
	6039	2403	7.30183	65	202	7.47846	7.29351	2.47%	35.2	105s
	6078	2436	7.38431	73	129	7.47846	7.29377	2.47%	36.3	110s
H	6080	2315				7.4540747	7.29377	2.15%	36.3	110s
H	6162	2232				7.4532591	7.29377	2.14%	36.4	112s
	6327	2263	7.34099	78	157	7.45326	7.29377	2.14%	37.0	115s

H	6337	2151			7.4389950	7.29377	1.95%	37.0	115s
	6664	2222	7.42110	86	25	7.43900	7.29408	1.95%	39.3	120s
H	6752	2126			7.4373450	7.29408	1.93%	40.1	122s
	6967	2210	7.31626	90	112	7.43735	7.29921	1.86%	41.4	125s
	7717	2336	7.33312	81	110	7.43735	7.30362	1.80%	43.4	130s
	8290	2520	7.32873	79	156	7.43735	7.30502	1.78%	44.2	135s
*	8927	2453		103		7.4357345	7.30645	1.74%	44.2	138s
	9167	2545	7.42233	103	28	7.43573	7.30753	1.72%	44.4	140s
H	9737	2412				7.4340845	7.30971	1.67%	44.9	144s
	9738	2461	infeasible	79		7.43408	7.30971	1.67%	44.9	145s
	10148	2384	7.40939	77	32	7.43408	7.31092	1.66%	45.5	150s
	10160	2392	7.40130	85	205	7.43408	7.31092	1.66%	45.4	155s
	10165	2396	7.31542	75	205	7.43408	7.31092	1.66%	45.4	160s
	10242	2446	7.33196	92	197	7.43408	7.31092	1.66%	46.4	165s
H	10276	2340				7.4101037	7.31092	1.34%	46.5	168s
	10295	2346	7.31092	84	153	7.41010	7.31092	1.34%	46.5	170s
	10455	2395	7.32437	87	196	7.41010	7.31092	1.34%	47.3	175s
*	10494	2282		106		7.3930222	7.31092	1.11%	47.3	175s
	10784	2331	7.31092	85	33	7.39302	7.31092	1.11%	48.0	180s
*	10832	2170		103		7.3727667	7.31092	0.84%	48.2	180s
	11185	2222	7.34114	87	105	7.37277	7.31092	0.84%	49.7	185s
	11845	2239	7.32844	94	182	7.37277	7.31092	0.84%	51.2	190s
	11857	2247	7.34964	103	213	7.37277	7.31092	0.84%	51.1	195s
	11869	2255	7.31987	90	212	7.37277	7.31092	0.84%	51.1	200s
	11873	2258	7.36537	97	212	7.37277	7.31092	0.84%	51.1	205s
	11925	2290	7.31092	100	166	7.37277	7.31092	0.84%	51.9	210s
	12054	2303	7.31609	102	204	7.37277	7.31092	0.84%	52.4	215s
	12235	2319	7.31092	100	122	7.37277	7.31092	0.84%	52.9	220s
	12448	2339	7.34775	108	30	7.37277	7.31092	0.84%	53.1	225s
	12832	2357	cutoff	112		7.37277	7.31092	0.84%	53.9	230s
	13248	2397	7.32710	105	185	7.37277	7.31302	0.81%	54.5	235s
	13996	2350	7.33186	103	193	7.37277	7.31678	0.76%	55.4	240s
	14657	2387	7.34147	105	188	7.37277	7.31875	0.73%	55.6	245s
	15580	2316	7.34110	110	72	7.37277	7.32378	0.66%	56.9	250s
	16440	2221	7.34526	124	50	7.37277	7.32640	0.63%	58.2	255s
	17024	2094	7.36744	115	192	7.37277	7.32877	0.60%	59.2	260s
	17890	1935	infeasible	111		7.37277	7.33208	0.55%	60.2	266s
	18699	2316	7.36785	103	60	7.37277	7.33326	0.54%	59.7	270s
	19887	3052	7.33880	144	29	7.37277	7.33390	0.53%	58.1	275s
	21409	3675	7.36082	125	45	7.37277	7.33496	0.51%	55.7	280s
	22246	4170	7.37131	130	14	7.37277	7.33513	0.51%	54.6	286s
	23827	4531	7.36993	152	31	7.37277	7.33578	0.50%	52.5	290s
	25408	5203	cutoff	121		7.37277	7.33638	0.49%	50.7	295s

27556	5864	cutoff	109		7.37277	7.33761	0.48%	48.5	300s
29015	6469	7.34504	158	10	7.37277	7.33844	0.47%	47.3	305s
30552	7004	7.36483	132	38	7.37277	7.33916	0.46%	46.1	310s
32203	7780	7.34125	129	32	7.37277	7.33979	0.45%	44.9	315s
32720	8047	cutoff	135		7.37277	7.33997	0.44%	44.5	341s
34188	8432	7.36047	131	9	7.37277	7.34056	0.44%	43.6	345s
35966	9078	cutoff	142		7.37277	7.34117	0.43%	42.5	350s
37538	9590	7.36456	125	36	7.37277	7.34170	0.42%	41.9	355s
39212	10098	7.35468	142	13	7.37277	7.34243	0.41%	41.2	360s
40718	10474	7.36113	127	22	7.37277	7.34326	0.40%	40.6	365s
42290	10956	7.34922	119	125	7.37277	7.34375	0.39%	40.1	371s
43836	11359	7.36513	118	34	7.37277	7.34432	0.39%	39.5	376s
44818	11588	7.34644	108	195	7.37277	7.34482	0.38%	39.3	381s
46159	12013	infeasible	110		7.37277	7.34537	0.37%	39.0	385s
47506	12332	7.36971	109	28	7.37277	7.34589	0.36%	38.7	390s
48288	12629	7.36068	145	8	7.37277	7.34622	0.36%	38.5	395s
50284	13209	7.35660	120	9	7.37277	7.34696	0.35%	38.0	401s
51561	13426	7.36488	114	36	7.37277	7.34741	0.34%	37.7	405s
53035	13782	7.36362	119	26	7.37277	7.34791	0.34%	37.2	410s
54634	14102	7.35759	125	30	7.37277	7.34848	0.33%	36.7	415s
56134	14541	7.36021	120	47	7.37277	7.34879	0.33%	36.3	420s
57881	14815	7.36939	135	16	7.37277	7.34931	0.32%	35.9	425s
59794	15237	7.35552	118	18	7.37277	7.34989	0.31%	35.3	430s
62016	15698	7.36927	143	10	7.37277	7.35047	0.30%	34.7	435s
63808	16187	cutoff	112		7.37277	7.35092	0.30%	34.2	441s
65176	16387	cutoff	138		7.37277	7.35129	0.29%	34.0	445s
67369	16604	cutoff	142		7.37277	7.35183	0.28%	33.5	450s
68929	16851	cutoff	137		7.37277	7.35230	0.28%	33.2	456s
70813	17029	infeasible	112		7.37277	7.35270	0.27%	32.8	460s
72596	17341	7.36308	113	22	7.37277	7.35309	0.27%	32.4	466s
73897	17421	7.35575	128	14	7.37277	7.35340	0.26%	32.2	470s
76116	17579	7.37022	107	216	7.37277	7.35392	0.26%	31.9	475s
78032	17665	cutoff	134		7.37277	7.35436	0.25%	31.5	480s
80287	17732	7.36353	123	41	7.37277	7.35486	0.24%	31.2	485s
82424	17669	7.35994	122	12	7.37277	7.35536	0.24%	30.9	490s
84279	17666	7.37172	121	20	7.37277	7.35580	0.23%	30.6	495s
86694	17724	7.36495	114	42	7.37277	7.35627	0.22%	30.2	500s
89139	17631	7.35972	133	9	7.37277	7.35678	0.22%	29.8	506s
90657	17474	7.36426	119	30	7.37277	7.35707	0.21%	29.6	510s
92288	17306	7.36600	118	6	7.37277	7.35748	0.21%	29.4	515s
94368	17167	7.36745	128	15	7.37277	7.35790	0.20%	29.2	520s
95740	17123	infeasible	124		7.37277	7.35814	0.20%	29.0	525s
97572	16876	7.37147	146	14	7.37277	7.35850	0.19%	28.8	530s

```
 99320 16586    7.37217  126    6    7.37277    7.35894  0.19%  28.6  535s
 101529 16315 infeasible  132         7.37277    7.35940  0.18%  28.3  540s
 103001 16102 infeasible  143         7.37277    7.35972  0.18%  28.2  545s
 104835 15769 infeasible  121         7.37277    7.36010  0.17%  28.0  551s
 107039 15312     cutoff  141         7.37277    7.36065  0.16%  27.7  555s
 108759 14799    7.36387  120   22    7.37277    7.36104  0.16%  27.6  560s
 110497 14285     cutoff  110         7.37277    7.36153  0.15%  27.4  565s
 112341 13661    7.36610  130    6    7.37277    7.36198  0.15%  27.2  570s
 114244 13134     cutoff  108         7.37277    7.36242  0.14%  27.0  576s
 116011 12535    7.37162  118   12    7.37277    7.36291  0.13%  26.9  580s
 117968 11574    7.36800  120    6    7.37277    7.36344  0.13%  26.7  586s
 119784 10739    7.36596  121   38    7.37277    7.36398  0.12%  26.5  590s
 121617  9696     cutoff  141         7.37277    7.36459  0.11%  26.3  595s
 123433  8651     cutoff  117         7.37277    7.36525  0.10%  26.2  600s
 125217  7596    7.36730  124   10    7.37277    7.36587  0.09%  26.0  605s
 127406  6083     cutoff  123         7.37277    7.36670  0.08%  25.8  610s
 128925  4912     cutoff  123         7.37277    7.36737  0.07%  25.7  615s
 130974  3751     cutoff  150         7.37277    7.36837  0.06%  25.5  620s
 132351  2109     cutoff  125         7.37277    7.36913  0.05%  25.4  625s
 135203   178    7.37193  126    6    7.37277    7.37136  0.02%  25.1  630s

Cutting planes:
  Learned: 22
  Gomory: 74
  Lift-and-project: 1
  Cover: 61
  Implied bound: 45
  MIR: 34
  Mixing: 1
  Flow cover: 241
  Inf proof: 48
  Zero half: 45
  RLT: 1

Explored 135366 nodes (3394417 simplex iterations) in 630.15 seconds (572.87 work units)
Thread count was 8 (of 8 available processors)

Solution count 10: 7.37277 7.37277 7.39302 ... 7.45407

Optimal solution found (tolerance 1.00e-04)
Best objective 7.372766731908e+00, best bound 7.372357568831e+00, gap 0.0055%

User-callback calls 338564, time in user-callback 5.70 sec
```


Cities
Tour


Optimal TSP Tour
Cities
Tour
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00