Colton Cunov
Stochastic Optimization Algorithms, Sep 2020
19940810-T732 (temporary), 940810-5378 (official)

Home problems, set 2

Problem 2.1, 5p, The traveling salesman problem (TSP)

a) *In the TSP, paths that start in different cities but run through the cities in the same order are equivalent (in the sense that the path length is the same). Furthermore, paths that go through a given sequence of cities in opposite order are also equivalent. Thus, for example, in a 5-city case, the paths (1, 2, 3, 4, 5) and (2, 3, 4, 5, 1) are equivalent, as are (1, 2, 3, 4, 5) and (5, 4, 3, 2, 1). Paths that are not equivalent are called distinct paths. How many distinct paths are there in the general case of N cities? Show clearly how you arrive at your answer!*

There are $N!$ possible non-distinct paths, since you have $N-0$ choices for the 1st city, $N-1$ choices for the 2nd city, $N-2$ for the 3rd, …, i.e. $N-i$ choices for the i^{th} city where $i=\{0,1,2,\dots,\mathrm{N}-1\}$. Multiplied together we have

$$\prod_{i=0}^{N-1} N-i = N!$$

- If, for every path, the starting city can be moved to end of the path and result in an equivalent path, as is the case for (1,2,3,4,5) ≈ (2,3,4,5,1), then we have $(N-1)!$ possible paths.
- If reversed patterns are equivalent, and each pattern has 1 reverse, then half of our patterns will be equivalent. So, we can divide by two to get

$$N_{distinct} = \frac{(N-1)!}{2}$$

b) *Write a GA (named GA21b.m) that can search for the shortest path between N cities, using permutation encoding (see p. 48 in the course book) for the path, i.e. an encoding method such that the chromosomes consist of lists of integers determining the indices of the cities…*

To run the GA with default parameters listed below, navigate to **Folder b** under **Folder 2.1** and execute **`GA21b`** in the command window. Tournament selection was used, and each gene was mutated with p_{mut}=0.03. As in the previous home problem set, `EvaluateIndividual.m` returns a fitness measure (taken as the inverse of the total path length) for a single individual, `TournamentSelect.m` executes tournament selection, `Mutate.m` executes mutation, and `InitializePopulation.m` generates a random population set.

N	50
Population size	200
Mutation probability	0.03
Tournament selection parameter	0.75
Tournament size	8
Number of generations	2500

c) *Ant colony optimization (ACO) is especially suitable for solving routing problems of the kind described above. Following the description in Chapter 4, implement an Ant System (AS) algorithm (Algorithm 4.1), for solving the same problem as in b).*

To execute the AS with default parameters shown below, navigate to **Folder c** under **Folder 2.1** and execute **`AntSystem`** in the command window. In this problem

- `InitializePheromoneLevels.m` creates an NxN matrix with all values = τ_0
- `GenerateDistanceMatrix.m` generates a symmetric distance matrix (where e.g. the value in row 5, column 12 is the distance from city 5 to city 12)
- `GetVisibility.m` outputs the inverse of the distance matrix
- `GetPathLength.m` calculates the total length of a given path
- `ComputeDeltaPheromoneLevels.m` computes $\Delta \tau_{ij}$
- `UpdatePheromoneLevels.m` updates the pheromone levels τ_{ij}
- `GeneratePath.m` generates a random path with probabilities $p(e_{ij}|S)$
- `CalculateDenominator.m` calculates the denominator $\sum_{v_l \notin L_{T(S)}} {\tau_{ij}}^{\alpha} {\eta_{ij}}^{\beta}$ for $p(e_{ij}|S)$
- `CalculateNumerator.m` calculates the numerator ${\tau_{ij}}^{\alpha} {\eta_{ij}}^{\beta}$ for $p(e_{ij}|S)$

Number of ants	50
α	1.0
β	3.5
ρ	0.7

d) *Write a simple program NNPathLengthCalculator.m that, after loading the city coordinates, does the following: (1) selects a random starting city; (2) generates the NN path, starting from that city; (3) computes the length of this path (including the final step, such that the tour ends at the city of origin).*

To find the Nearest Neighbor path length, execute

```
cityLocation = LoadCityLocations();
NNPathLengthCalculator(cityLocation)
```

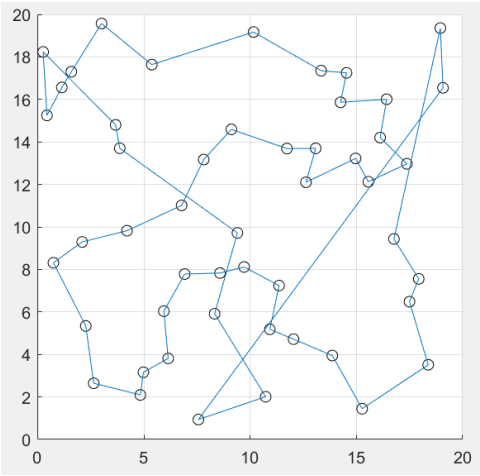
in the command window. Note that there is a total of N=number of cities (50) possible outputs to this function, since the starting city is chosen at random. A sample result, along with a plot of the path (not included as part of the function) is shown in the next section.

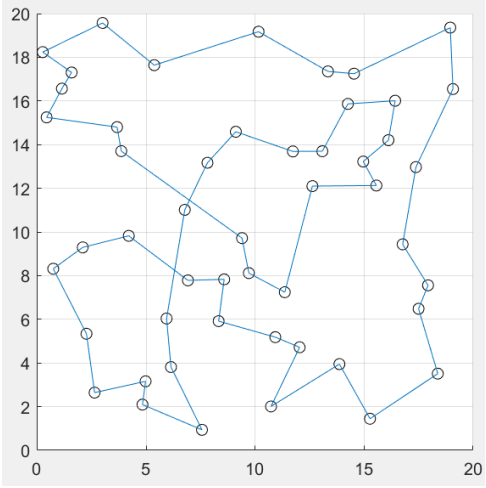
e) *Next, make a long run with each of the two algorithms (GA and ACO) described above. In your report, plot the shortest path found with each of the two algorithms and specify the length of each of these paths. Specify also the length of the nearest-neighbour path from part d).*

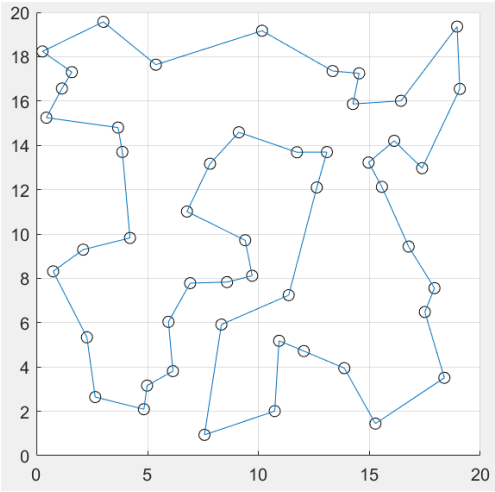
The shortest path was found with Ant System and can be found in the file **`BestResultFound.m`** under **Folder 2.1**.

Best Paths Found

Nearest Neighbor	Genetic Algorithm	Ant System
145.9045	**133.1328**	**122.8053**







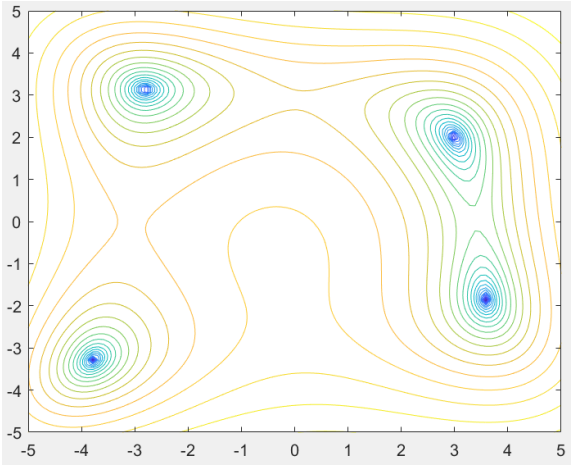
Colton Cunov
Stochastic Optimization Algorithms, Sep 2020
19940810-T732 (temporary), 940810-5378 (official)

Problem 2.2, 2p, Particle swarm optimization

Start by implementing a standard PSO algorithm (as described in Chapter 5) in Matlab. Note: The standard PSO algorithm should include the (varying) inertia weight, see p. 128 and Eq. (5.20) in the course book! Remember to follow the coding standard and to place separate Matlab functions in separate files. Next, use the contour command in Matlab to determine the number of minima of the function

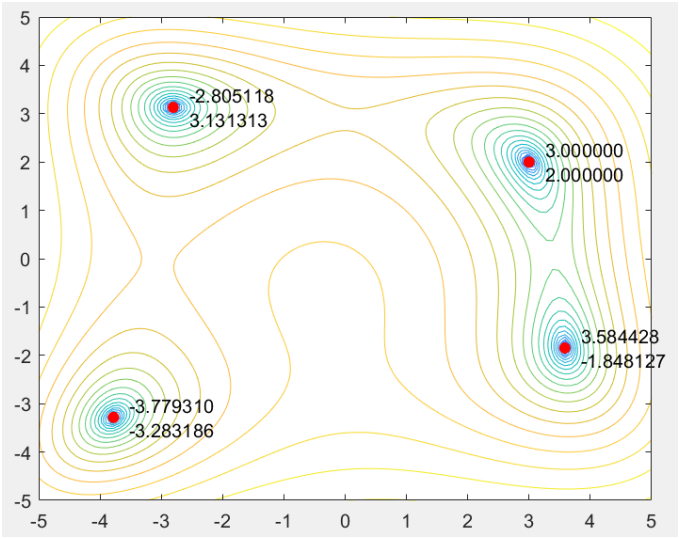
$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

over the range $(x, y) \in [-5,5]$. *You should include the contour plot, with the minima clearly identified, in your report.*



Plotting the function $\log(.01 + f(x,y))$ yields the contour plot to the right. There are (4) local minima over the specified range, shown in blue.

Next, use your PSO (named PSO22.m) to find the exact location of all the local minima (as well as the corresponding function values) of the function f(x, y). In your report, you should provide a table with (x, y, f) for the minima.



x	*y*	*f(x,y)*
3.584428	-1.848127	0.0
-3.779310	-3.283186	0.0
3.000000	2.000000	0.0
-2.805118	3.131313	0.0

To run the algorithm with default parameters shown below, navigate to **Folder 2.2** execute **`PSO22`** in the command window. It will run until four local minima are found and print the plot shown to the left.

Number of boids (N)	25
Search space	x=[-5 5], y=[-5 5]
α	0.5
Time step	1
c1	2
c2	2
Max velocity	3
Starting inertia	1.4
β	0.95
Lower inertia bound	0.35
Number of iterations	10,000

- `InitializePositions.m` assigns N boids to random positions in the specified range
- `InitializeVelocities.m` assigns velocities to N boids according to
$$v_{ij} = \frac{\alpha}{\Delta t}\left(-\frac{x_{max}-x_{min}}{2} + r(x_{max} - x_{min})\right)$$
- `EvaluateBoid.m` returns $f(x,y)$ for a particular boid
- `UpdatePositions.m` updates positions for all boids
- `UpdateVelocities.m` updates velocities for all boids
- `ContourPlot.m` generates a contour plot

Colton Cunov
Stochastic Optimization Algorithms, Sep 2020
19940810-T732 (temporary), 940810-5378 (official)

Problem 2.3, 4p, Optimization of braking systems

In this problem you will develop an intelligent system for braking heavy-duty trucks during complicated descents (see pp. 83-86 in the course book). Such trucks are equipped with several braking systems, including the ordinary pedal brakes (also called foundation brakes), engine brakes, retarder etc. In order for the (foundation) brakes to work properly, the brake discs must never be overheated. Here, we will use a strongly simplified model of a truck, which will now be described…

…Write a program that implements the truck model described above.

The main script is called `Main.m` in **Folder 2.3**. To run the algorithm with the parameters defined as below, execute `Main` in the command window.

Problem-specific Parameters

Gear ratio	[7, 5, 4, 3, 2.5, 2, 1.6, 1.4, 1.2, 1]
Shift request thresholds	[.3 .7]
Mass	20000 (kg)
τ	30 (s)
C_h	40 (K)
$T_{ambient}$	283 (m/s^2)
g	9.80665 (N)
C_b	3000 (m/s)
$[V_{min}, V_{max}]$	[1, 25] (m/s)
$T_{brake,max}$	750 (K)
L	1000 (m)
α_{max}	10 (degrees)
Time step	0.2 (s)
$V_{t=0}$	20 (m/s)
$X_{t=0}$	0 (m)
$T_{brake,t=0}$	500 (K)
$Gear_{t=0}$	7
Minimum time b/n shifts	2 (s)

Genetic Algorithm Parameters

Population size	100
Creep mutation probability	0.05
Creep mutation range	0.25
Ordinary mutation probability	0.05
Tournament selection parameter	0.75
Tournament size	5
Number of generations	500
Crossover probability	0.15

Neural Network Parameters

Number of neurons	8
Variable range	[-5, 5]
Weight initialization range	[-1, 1]
Shift request thresholds	[.3, .7]

The fitness for slope *i* was taken as the average velocity multiplied by the total distance covered before termination, however a 20% fitness penalty was placed on unfinished runs,
$F_i = \begin{cases} .8 * \bar{v}_i D_i & if\ D_i < 1000 \\ \bar{v}_i D_i & otherwise \end{cases}$. The total fitness was taken as the average fitness over all slopes.

The population was initialized with weights randomly assigned using a uniform distribution on the interval [-1,1] and all biases = 0.

Colton Cunov
Stochastic Optimization Algorithms, Sep 2020
19940810-T732 (temporary), 940810-5378 (official)

Genetic Algorithm Functions

- `InitializePopulation.m` initializes the population on the weight initialization range
- `EvaluateIndividual.m` evalutes the fitness of a single individual over a specified dataset
- `TournamentSelect.m` executes tournament selection
- `Cross.m` performs crossover
- `CreepMutate.m` randomly mutates genes over a small window around their current value
- `OrdinaryMutate.m` randomly mutates genes over the allowed variable range

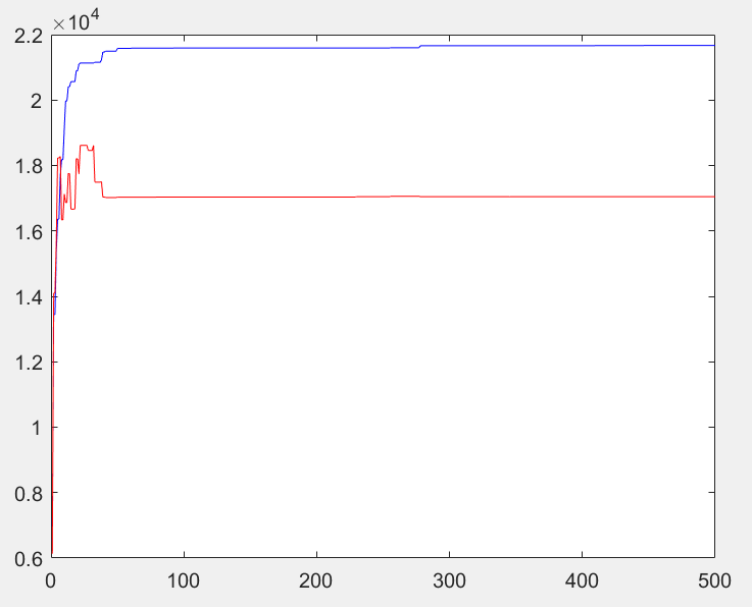
Neural Network Functions

- `CalculateRatios.m` calculates the ratios of velocity, alpha, and brake temperature to their maximum values
- `ANN.m` executes the neural network and returns two outputs (pedal pressure and shift request)
- `DecodeChromosome.m` decodes the chromosome array into weight matrices

Other Functions

- `GetBrakeForce.m` returns the braking force
- `GetEngineBrakeForce.m` returns the engine braking force
- `GetGravitationalForce.m` returns the gravitational force
- `CalculateAcceleration.m` returns the resulting accelerational force
- `UpdateVelocity.m` updates the velocity
- `UpdatePosition.m` updates the position
- `UpdateBrakeTemp.m` updates the brake temperature
- `LoadBestWeights.m` loads the best weights found during testing
- `DecodeShiftRequest.m` decodes the shift request variable (on the range [0, 1]) into a shift action {+1, -1}
- `LoadProblemSpecificParameters.m` returns a struct of parameter values

To the right is a plot of maximum fitness value in the population obtained on the training and validation sets.



Using holdout validation, the training was stopped at the 29th generation.

To test the best network on an arbitrary slope, run the program `TestProgram.m`. It will load the best network via the program `LoadBestWeights.m`, run the slope specified by the variables iDataSet and iSlope until the end of the track is reached or the constraints are violated, and output a set of subplots comprising of alpha, pedal pressure,

gear, velocity, and brake temp as a function of horizontal distance travelled

Problem 2.4, 4p, Function fitting using LGP

Consider a case in which a data series has been generated from a function of the form

$$g(x) = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_p x^p}{b_0 + b_1 x + b_2 x^2 + \cdots + b_q x^q}$$

The values of p and q, as well as the constants ai and bi are unknown, and should be inferred from the data series using LGP. Start by writing a general LGP program (called LGP24.m), with M variable registers, N constant registers, and the operator set {+, −, ×, /}. The program should evolve linear chromosomes using tournament selection, two-point crossover (see pp. 76-77 in the book), and mutations…

The general LGP program is in the **`LGP24.m`** file, which requires no inputs to run. A population of 100 individuals were initialized on a uniform random distribution with instruction lengths varying from 3 to 10.

M=4 variable registers, N=1 constant registers {-.8}, and the operator set {+,-,*,/} were used. The chromosomes evolved over G=10,000 generations using

- Tournament selection with $p_{tour} = 0.75$ and $N_{tour} = 5$
- Two-point crossover with $p_{cross} = 0.2$ where crossover points were chosen randomly on a uniform distribution
- Mutation with a decaying mutation rate dependent on the size of the chromosome $p_{mut} = \frac{d}{L}\frac{G-g}{G}$ where L is the length of the chromosome, the generation number $g = 1,2,\ldots,G$, and d is a constant (chosen as 3). Each gene was mutated with P(mutation) = p_{mut} to a value in the appropriate range.

Elitism was implemented with single replacement, and a fitness penalty was imposed on chromosomes with length L > 120 (or 30 instructions) according to $f_{penalized} = f * c^{(L-120)}$ where c = 0.9. The fitness f was taken as the inverse of the root mean square error.

To run the program with default parameters shown below, navigate to **Folder 2.4** and execute **`LGP24`** in the command window.

Number of variable registers	4
Constant registers	{-0.8}
cMax	100,000
Number of generations	10,000
Population size	100
Tournament selection parameter	0.75
Tournament size	5
Crossover probability	0.2
Initial mutation constant (*d*)	3
Range of initial chromosome lengths	[3 10]
Upper limit of chromosome length	30
Penalty for exceeding upper limit (*c*)	0.9

As in the previous assignments, `Cross.m` implements crossover, `Mutate.m` implements mutation, `TournamentSelect.m` implements tournament selection, `InitializePopulation.m` generates the initial population, and `EvaluateIndividual.m` returns a fitness value for a single individual. The file `ExecuteInstructions.m` is used to calculate the new value for an instruction, i.e. it outputs a value based on the operator and two operands.

The test program `TestLGPChromosome.m` requires no inputs and

- loads the best chromosome found from the LGP program (located in `BestChromosome.m`)
- decodes the chromosome into a human-readable function *g(x)* via the program `GenerateFinalFunction.m`
- plots the original function data y_k and the outputs obtained by $g(x_k)$
- outputs the root mean square error and the function *g(x)*.

The program obtained $g(x) = \frac{x^3 - x^2 + 1}{x^4 - x^2 + 1}$ with a root mean square error below $3 * 10^{-9}$