

**FIRST STEPS****Contents**

<b>1. General Information.....</b>	<b>2</b>
1.1. About.....	2
1.2. Authors and references .....	2
<b>2. How to Start.....</b>	<b>3</b>
2.1. Defining a solution pattern .....	5
2.2. Creating a population of solution candidates .....	6
2.3. Creating a genetic algorithm instance .....	7
2.4. Organizing an evolving loop.....	9
2.5. Composing a fitness function.....	11
2.6. Inspecting results.....	14
<b>3. Samples.....</b>	<b>16</b>
3.1. Evo. DiagonalProblem sample .....	16
3.2. Evo. AckleyFunction sample.....	18
3.3. Evo. TravellingSalesmanProblem sample.....	20

## 1. General Information

### 1.1. About

**Evo** is a custom open-source package designed to be used in Dynamo environment (version 1.3.3. or newer). So far includes C#-coded nodes responsible for solving single-objective optimization problems using Genetic Algorithms (GA). The package is suitable for optimizing multi-variable number-based problems as well as combinatorial ones, e.g. [Travelling Salesman Problem](#)<sup>1</sup>.

The document presents basic information about preparing visual scripts using the package. It also brings elementary terms and methodologies used in the solution as well as describes samples enclosed to the package<sup>1</sup>. Integral part of the specification are separate documents: *Evo Specification. Crossovers.pdf*, *Evo Specification. Mutations.pdf* and *Evo Specification. Selections.pdf* including theory background regarding basic GA operators: crossover, mutation and selection methodologies.

### 1.2. Authors and references

The package was created by [Marcin Jasiński](#)<sup>1</sup>, a PhD candidate at the Silesian University of Technology, Faculty of Civil Engineering, Department of Mechanics and Bridges, Gliwice, Poland. The source of the package is available at [Evo](#)<sup>1</sup> GitHub repository. Contact: [marcin.jasinski@polsl.pl](mailto:marcin.jasinski@polsl.pl).



The main part of the solution is based on [GeneticSharp](#)<sup>1</sup> by **Diego Giacomelli** – a fast, extensible, multi-platform and multithreading C# Genetic Algorithm library that simplifies the development of applications using Genetic Algorithms (GA). License: MIT.




---

<sup>1</sup> Can be found in: %APPDATA%\Dynamo\Dynamo Core [or Dynamo Revit]\1.3 [or 2.0]\packages\Evo\extra

## 2. How to Start

All the package nodes are included in **Evo** → **GeneticAlgorithms** category visible in Dynamo libraries panel. The category is divided into several subcategories:

Category	Description
<p>Basic</p> 	<p>A group of nodes responsible for crucial application of the genetic algorithm solver. In general, each Dynamo script utilizing the package should contain at least one of each: <b>Basic.CreateGeneticAlgorithm</b>, <b>Evo.Init</b>, <b>Evo.ContinueWhile</b> and <b>Evo.LoopBody</b> nodes. It should also contain at least one <b>LoopWhile</b> node which is package-independent Dynamo node available under BuiltIn category.</p>
<p>Chromosomes</p> 	<p>A group of nodes responsible for definition of a solution candidate pattern, i.e. how many variables are there in the optimization problem, what are their minimum and maximum values, what is their required precision and so on.</p>
<p>Crossovers</p> 	<p>Crossover, besides selection and mutation, is one of the basic GA operators. Crossover is responsible for mating the best parents in the input generation.. Crossover nodes define objects that are send to the <b>Basic.CreateGeneticAlgorithm</b> node.</p>
<p>Mutations</p> 	<p>Mutation, besides selection and crossover, is one of the basic GA operators. Mutation is responsible for slight modification of randomly chosen offsprings produced by the crossover. Mutation reintroduces genetic diversity back into the population and prevents from catching local optima, right after converging pressure imposed by crossover. Mutation nodes define objects that are send to the <b>Basic.CreateGeneticAlgorithm</b> node.</p>

Category	Description
<b>PopulationStrategies</b> 	<p>A small group of nodes that define the way in which generations are stored within a genetic algorithm instance. The instance is created by <b>Basic.CreateGeneticAlgorithm</b> node and iterated by <b>LoopWhile</b> built-in Dynamo node. Each iteration produces new generation. In the case of a huge problem, a significant number of generations may be produced. The strategy nodes define if all generations should be stored in the solution or only selected ones.</p>
<b>Populations</b> 	<p>A group of nodes responsible for operations on populations and their generations. A population is a set of generations. A generation is a set of candidate solutions. At the beginning, a population is generated using <b>Populations.GeneratePopulation</b> node, basing on the defined chromosome pattern. The newly generated population includes initial generation of randomly obtained solution candidates. The solver creates new generations in the population that can be then accessed by other nodes from this category.</p>
<b>Selections</b> 	<p>Selection, besides crossover and mutation, is one of the basic GA operators. Selection is responsible for picking individuals from the most recent generation to so called parents' set. The parents are then mated by one of the crossover operators and produce offspring. Selection methodologies try to select the best candidates but also try to maintain a certain diversity level in the parents' set. Selection nodes define objects that are send to the <b>Basic.CreateGeneticAlgorithm</b> node.</p>
<b>Terminations</b> 	<p>A group of nodes defining specific conditions that are required to stop a running evolving processes.</p>

## 2.1. Defining a solution pattern

First of all, the algorithm should know a kind of the problem to solve. If the optimization problem is a single-objective function of multiple variables, that can be expressed mathematically, one of the **Chromosomes.CreateFloatingPointChromosome** nodes should be used.

If the problem is a single-variable optimization, at least minimum and maximum values of the variable should be given by connecting Number nodes to the **Chromosomes.CreateFloatingPointChromosome** node:

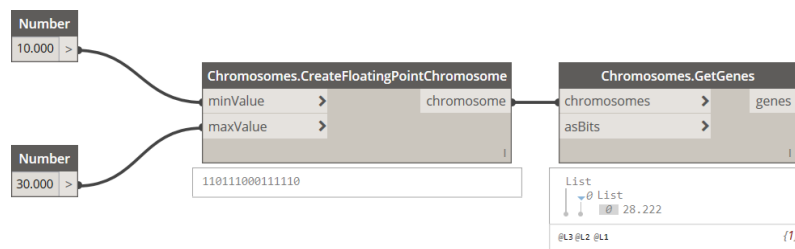


Fig. 1. Creating a floating point chromosome for a single-variable function.

As seen above, output of the **Chromosomes.CreateFloatingPointChromosome** node has a form of binary representation string that, after decoding, is equal to 28.222. Decoding may be done by **GetGenes** or **GetDecodedGenes** nodes from the Chromosomes category. The encoded chromosome is an exemplary solution candidate that will be used to generate a bigger population of similar candidates when passed to the **Populations.GeneratePopulation** node (see: chapter 2.2). One can also provide required precision of the variable (fraction digits, 3 by default) and, if desired, number of bits (length of the binary string). Please note that negative numbers generate long binary representations and can lower convergence rate of the solution (see: sample 3.2).

If the problem is a multi-variable optimization, instead of single values, lists of the values should be passed to the **Chromosomes.CreateFloatingPointChromosome** nodes, i.e. list of minimum values, list of maximum values and, if needed, list of fraction digits and list of binary representation lengths:

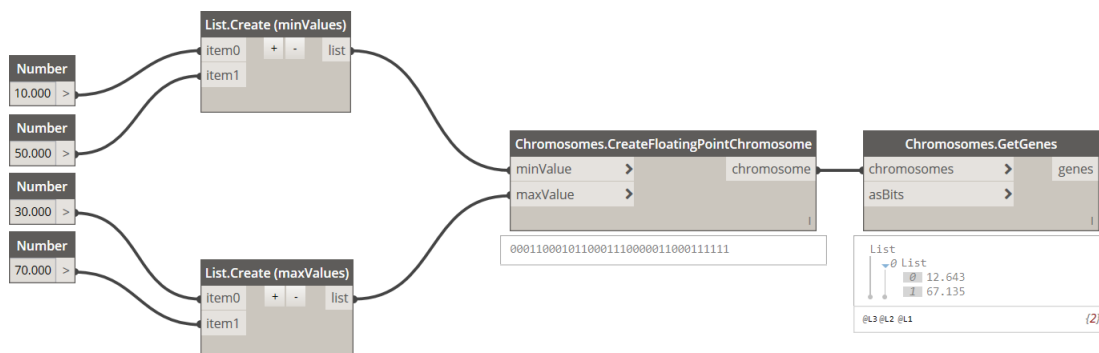


Fig. 2. Creating a floating point chromosome for a multi-variable function.

If the problem is a combinatorial optimization, e.g. Travelling Salesman Problem (TSP), it is required to obtain a chromosome in a form of an array of unique elements (numbers). In such the case the dedicated **Chromosomes.CreateOrderedChromosome** node can be used. The node takes a number of elements as an input. The output has a form a numerical string translated to a list of unique numbers when decoded (see: sample 3.3).

## 2.2. Creating a population of solution candidates

The exemplary solution candidate created in chapter 2.1. can be then used to generate the whole population of other candidates based on the same constraints (minimum and maximum values, required precisions and so on). To do so, one of the **Populations.CreatePopulation** nodes should be used:

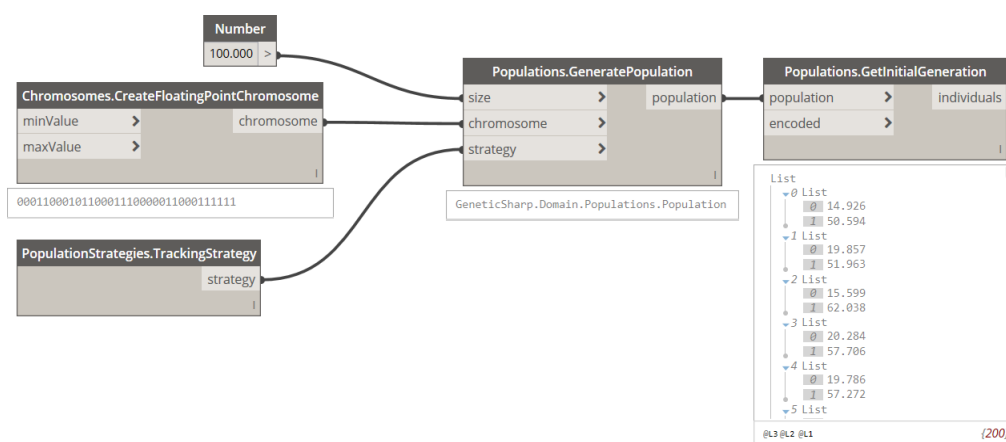


Fig. 3. Creating a new population.

The required inputs are: size of the population (number of solution candidates) and the exemplary chromosome. The created population can be also supplemented with a strategy defined by one of the **PopulationStrategies** nodes. So far there are two population strategies available: **TrackingStrategy** that records all individuals of all generations created during optimum search (in case of complex problems number of generations may be significant); and **PerformanceStrategy** that records only a given number of most recent generations (older generations, including initial generations are overwritten). If no strategy is specified, **PerformanceStrategy** is taken by default with 10 most recent generations recorded.

Creating a population generates also an initial generation. If not yet overwritten by the strategy, it can be inspected by the **Population.GetInitialGeneration** node. In the Fig. 3, a new population of size 100 was created. It utilizes the chromosome defined in the Fig. 2, hence 200 numbers are encoded in the population (100 solution candidates – 100 sets of two variables per each set).

### 2.3. Creating a genetic algorithm instance

The population from the previous point, so far including the initial generation only, should be passed to the **Basic.CreateGeneticAlgorithm** node. The node is used to define what population should be evolved and which operators should be used.

There are three main GA operators, in turn: selection, crossover and mutation. Selection is used for inspecting the input generation and selecting a portion of most fitting individuals. Usually, selection methodologies are also designed to ensure that a certain diversity level in the selected set is maintained so the algorithm is less likely to converge to local optima. Both the selection method and size of the selection set are inputs of the **Basic.CreateGeneticAlgorithm** node. The selection size can be lower or equal to the population size defined in the previous point. Selected elements are passed to crossover and mutation. The remaining slots in the new generation are filled with unchanged elements from the input population, without crossover and mutation imposed. So far elitist reinsertion is used, meaning that the best input individuals will take the free slots in the generation.

The selection method is an object created by one of the nodes included in the **Selections** category. So far four selection methodologies are implemented in the solution, described in *Evo Specification. Selections.pdf*.

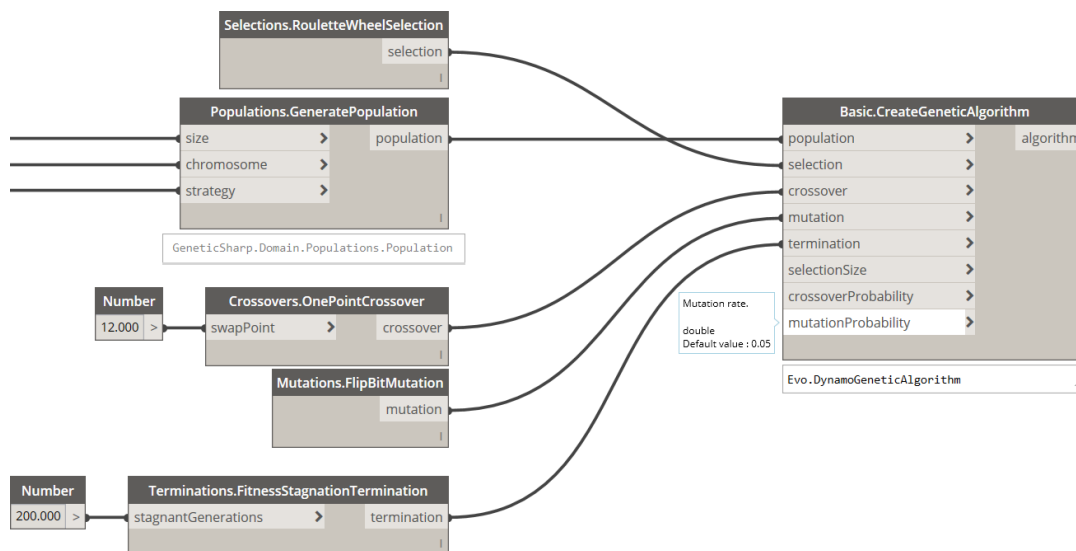


Fig. 4. *Basic.CreateGeneticAlgorithm* node with several inputs connected.

Selected individuals are called parents. In the crossover, parents are matched with each other, producing new individuals called offsprings. Usually one pair of the parents, randomly selected from the parents' set, produce two new offsprings. It should be however noted that there are several crossover methods that are exceptions to this principle. Structure of the offspring consists of its parents'

structures, combined in a specific way. Still, not all parents take part in the crossover, even if passed to the parents' set. Some portion of the parents is directly taken to the offsprings' set with no change in their structures. The crossover probability is one of the GA parameters, set to 0.75 by default. It is ensured that size of the final offsprings' set is equal to size of the parents' set. So far twelve crossover methodologies are implemented, described in *Evo Specification. Crossovers.pdf*. These are objects returned by nodes of the **Crossovers** category.

Finally, the offsprings' set is affected by the mutation operator. In (Konak et al., 2006), it is said that “[...] *crossover leads the population to converge by making the chromosomes in the population alike. Mutation reintroduces genetic diversity back into the population and assists the search escape from local optima.*”. Indeed, newer generations in the population are characterized by relatively low diversity. The point is to make sure that the fitness function values evaluated from such the generation is the global optimum, not a local one. The mutation operator is responsible for slight modification of an offspring, usually at gene level. There are seven mutation operators implemented so far, each one working in a specific way, described in details in *Evo Specification. Mutations.pdf*. Not all offsprings are however mutated – the mutation probability is one of GA parameters and is relatively low compared to the crossover probability, usually between 0.01 and 0.05. The value of 0.05 is used by default. The desired mutation method should be passed to the **Basic.CreateGeneticAlgorithm** node, returned by one of the nodes grouped in the **Mutations** category.

Finally, the termination can be defined. So far there are four different termination conditions that define the break point in the evolving loop. These are included in the **Terminations** category and can be combined by two additional operator nodes: *AndTerminations* and *OrTerminations*. E.g. it is possible to break the solution search process if a given number of generations was produced, when a given number of newly produced generations do not produce any better individuals, when evolving time is exceeded or several conditions combined.



## 2.4. Organizing an evolving loop

Dynamo specificity requires organizing a set of nodes that will iterate through the whole evolving process. A single iteration includes selection, crossover and mutation of individuals and these are the role of the **Basic.EvolveGeneration** node. Basing on the sequence, a new generation is produced that is, by nature, fitter than the previous one. Because after each iteration a fitness function results for new individuals must be evaluated, the function body must be passed somewhere as a parameter. In textural programming, procedures or functions taking other functions as parameters are called higher-order methods and these are hard to implement due to the specific way in which functions are encoded in Dynamo.

Therefore, a Dynamo built-in **LoopWhile** node will be utilized. The node is not a part of the package but can be found in the Dynamo **BuiltIn** category. If you never used the LoopWhile node yet, some basic information can be found [here](#). As you can read, the node takes three inputs: *init*, *continueWhile* and *loopBody*. The last two are the function-type inputs. In Dynamo, it is easy to distinguish between nodes that return functions and nodes that return results of these functions. If a node does not receive all required inputs, its title bar has a light grey color (see Fig. 5). Such the node returns a function object that can be used by some other nodes in Dynamo library, e.g. List.Map. If all required inputs are passed to the node, its title bar is brown, hence returning function results. Note that some nodes inputs may stay unconnected yet still the node title bar is dark, what indicates that unhandled inputs take default values, imposed programmatically. It is important to connect light greyed nodes to *continueWhile* and *loopBody* inputs of the LoopWhile node. Still, a browned node should be passed to the LoopWhile as the *init* input.

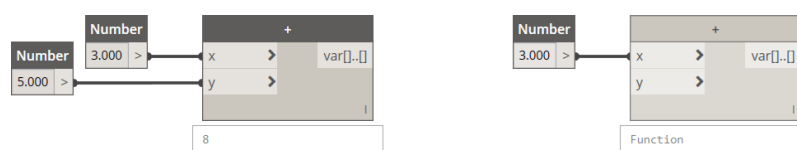


Fig. 5. Nodes returning results of functions (left) and function objects (right).

Integral part of the Evo package are three custom nodes that handle all the three LoopWhile inputs and make using the LoopWhile node more intuitive. These nodes are included in the **Basic** category: **Evo.Init**, **Evo.ContinueWhile** and **Evo.LoopBody** for *init*, *conitnueWhile* and *loopBody* LoopWhile inputs, respectively. The nodes are clusters of other nodes, including standard Dynamo ones. Such the structures are saved as separated DYF files and are indicated by slightly modified outfit among other nodes in the network. To inspect them, simply put one of the nodes somewhere in the work area, click right mouse button on the node and then, from the context menu: *Edit Custom Node*. It may be noticed

that the **Evo.LoopBody** custom node includes the **Evo.EvolveGeneration** node that is responsible for creating and modifying new generations in the evolving search process.

An exemplary arrangement of the **LoopWhile** and **Evo** nodes is presented in the figure below. Make sure there is nothing passed to *init* inputs of **Evo.ContinueWhile** and **Evo.LoopBody** nodes. The *keepEvolving* input of **Evo.init** node may stay unhandled as it takes *true* by default. If you want to disable evolving process at start, just pass *false* value to the *keepEvolving* input and change it when desired.

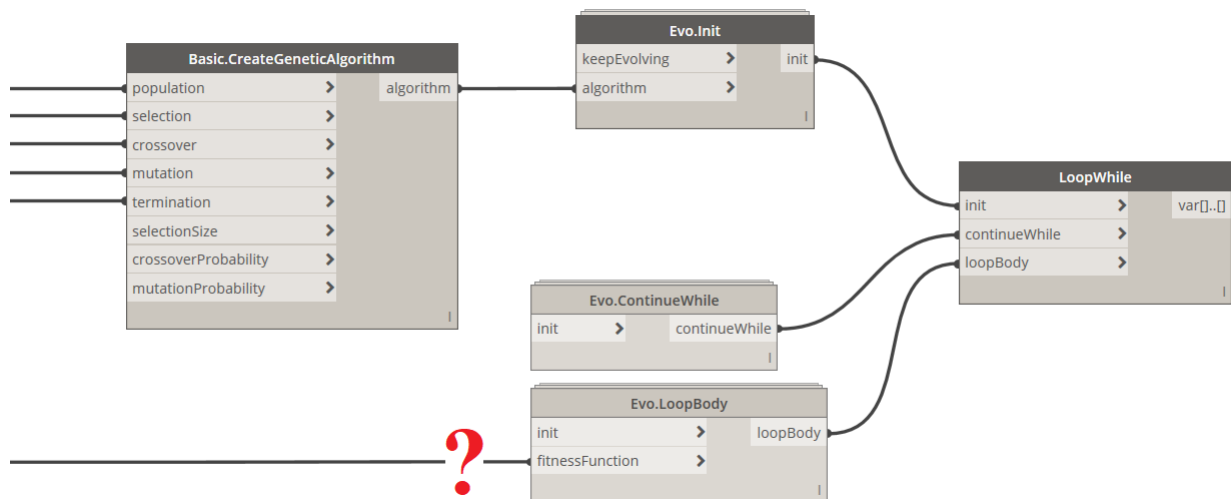


Fig. 6. Exemplary arrangement of **LoopWhile** and **Evo** nodes.

Last but not least is the *fitnessFunction* input of the **Evo.LoopBody** node. This is actually the key part of the optimization problem, that defines the function to optimize. Such the function is called a fitness function. Basic rules regarding composing the fitness function are given in the next chapter.

## 2.5. Composing a fitness function

The fitness function defines a problem to resolve. There are no special limits as for formulas and operations performed inside the functions. These may be mathematical or logical operations, using numbers, strings, geometries and any other custom types or utilizing outer environments and datasets, e.g. text files, excel sheets, Revit or even other platforms, e.g. Robot, if dedicated packages handling them are used. Note that in case of complex problems, utilizing external sources and environments, evolving time may be significant.

There are several rules that should be obeyed when composing a fitness function:

1. All of the operations inside the function should be composed inside a custom node (saved as external DYF file): *File* → *New* → *Custom Node*,
2. Keep in mind the fitness function takes a list of input variables, operates on them and returns a list of results. If a single-variable optimization is the goal, there is no special concerns regarding organizing the initial operations performed by the function – just a list of  $n$  variables is taken and a list of  $n$  results is evaluated.

However, if the optimization is a multi-variable problem, the fitness function should take a list of  $n$  **sets** of variables and return a list of  $n$  results – one numerical result per one set of variables. Each set can contain numerous single variables, hence prior to process the variables, their extraction out of the set is required. It can be noticed in one of the samples files **Evo. Diagonal Problem.dyn** and its custom DYF node called **DistanceBetweenPoints.dyf**<sup>2</sup>:

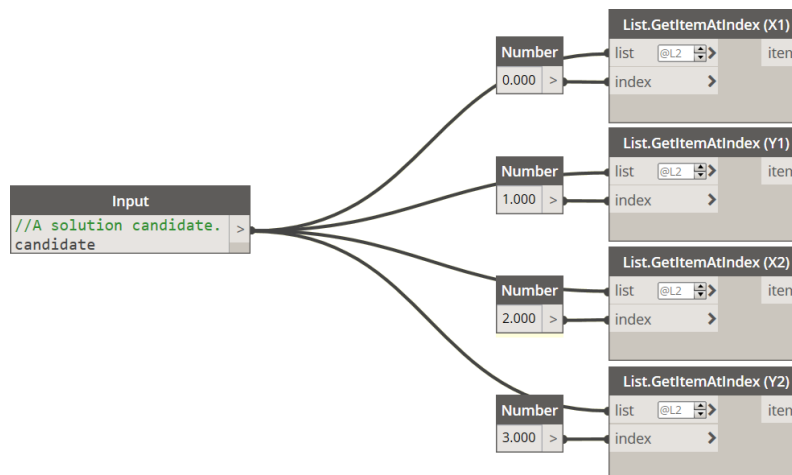


Fig. 7. Extraction of variables from a set of variables passed to the input of a fitness function custom node.

<sup>2</sup> Can be found in: %APPDATA%\Dynamo\Dynamo Core [or Dynamo Revit]\1.3 [or 2.0]\packages\Evo\dyf

In this particular case, an Euclidean distance between two points is calculated. Each point is defined by two numbers: X and Y coordinates for the first point and X and Y coordinates for the second point. Therefore, a four-variable problem is analyzed.

The input *candidate* may be a set of four variables or just a list of such the sets – in fact it does not matter as looping through the list of sets is handled automatically by Dynamo. Our concern is to take care of the four variables inside a single set passed from the *candidate* input node. The input and output nodes are available from standard Dynamo library, under **Input** category. The custom node we create can have several inputs and several outputs however **the whole generation of solution candidates** (in this case the whole list of four-variable sets) **should be passed to one and only one input**. Other inputs, if introduced, can be used to pass constant values to the fitness function that are not the optimization subject but are used to calculate the fitness function (e.g. a list of coordinates of cities in TravellingSalesmanProblem.dyf node).

As you can see in the Fig. 7, four coordinates are extracted out of a *candidate* using built-in *List.GetItemAtIndex* nodes. Note the levelling feature was used for their *list* inputs, meaning that if a list of sets is passed to the fitness function, *List.GetItemAtIndex* does **not** return first, second, third and fourth set, but first, second, third and fourth element from **inside** of the set. To do so, just click on the > icon to the right of the input title and tick the *Use Levels* option:

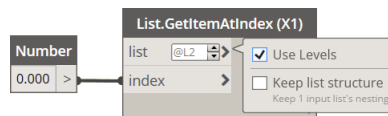


Fig. 8. Nodes levelling.

3. The custom node, for each set of variables should return one numerical value that is the fitness of the set. The final value obtained by the function operations should be passed to the **Output** node which is a built-in Dynamo node. The custom node can also calculate and return auxiliary results however only one output should be passed to the *fitnessFunction* input of the **Evo.** **LoopBody** node. The *fitnessFunction* input accepts only a function object so make sure nothing is connected to your custom node when passing it to the **Evo.** **LoopBody**:

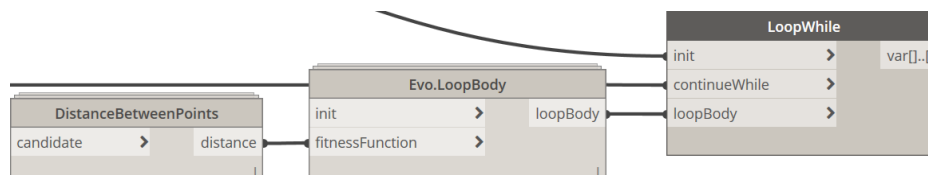


Fig. 9. A custom node defining a fitness function passed to the *Evo.LoopBody* node.

4. The searched global optimum is the **maximized fitness function** – the greater the result of the fitness function, the better candidate solution. Therefore, if your objective consist of function minimization, it is recommended to multiply final function result by  $-1$  right before passing it to the Output in the custom node, or – even better – using a function normalizing the fitness.
5. Errors in fitness function evaluation can lead to incorrect optima found or even termination of the whole evolving process. Therefore, it is recommended to test the fitness custom node before starting the genetic algorithm. Examples of such of the tests are presented in most of sample files, e.g. calculating distance between coordinates:

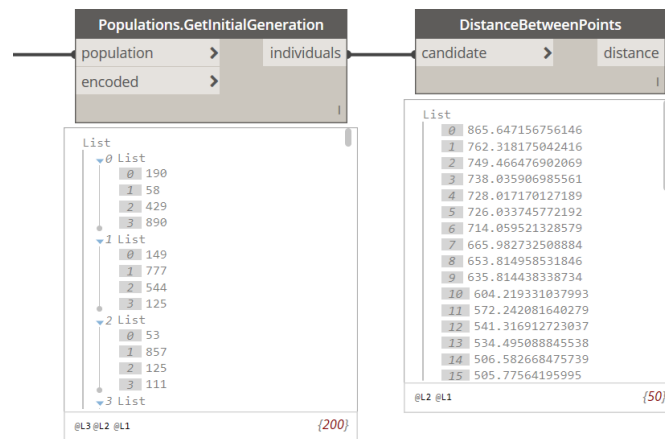


Fig. 10. Inspecting values returned by a fitness function custom node.

## 2.6. Inspecting results

Once the evolving process was finished, its results can be inspected. Note the **LoopWhile** node returns a list of two elements. First element is the *false* value meaning that the processing has finished and no further generations are created.

The second element is the genetic algorithm instance – the same we created in chapter 2.3, but now supplemented with new generations. Note that if **PopulationStrategies.PerformanceStrategy** was used as the population strategy, older populations, including the initial one may be overwritten and, from now on, inaccessible. If you want to store all generations that appear during the whole evolving process, just use **PopulationStrategies.TrackingStrategy**.

First of all the genetic algorithm instance must be taken from the LoopWhile output, preferably using **List.LastItem** available in the standard Dynamo library. Then, the population should be retrieved from the genetic algorithm instance, using the **Populations.GetPopulation** node. From the population, inner generations may be extracted. It is possible to retrieve elements from the most recent generation, nth generation or all elements of progressive generations (i.e. producing better and better fitness values).

In the example below, the **Populations.GetCurrentGeneration** node was used. The node returns list of solution candidates in the recently evolved generation. To extract the best solution (best chromosome), use the **BestChromosome** node from the **Chromosomes** category:

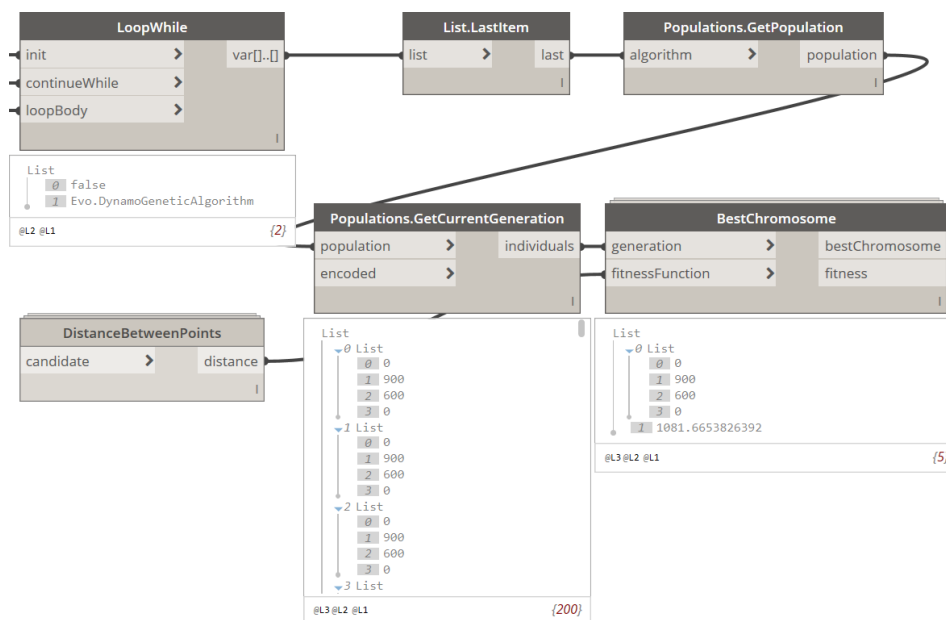


Fig. 11. Inspecting best solution of the optimization problem.

Please note that genetic algorithms are classified as metaheuristic techniques. Therefore, by definition, results obtained by such the techniques do not guarantee to be optimal, but instead sufficient for reaching an immediate goal. The obtained result is as close to the real global optimum as possible, however, due to stochastic methods, can slightly differ from results noted in previous or further runs. In most cases optima obtained from genetic algorithms are sufficient from the engineering point of view, as long as correct parameters and operators are used.

A large number of genetic algorithm evaluations were carried out in the research field. A lot of them include assessment of operators and parameters subjected to the given type of problem. The other ones suggest new selection, crossover or mutation methods to be introduced. The numerous implementations in this package cover these proposals. In further versions of the package, additional operators and features may be expected, including multi-objective optimization techniques.

### 3. Samples

This chapter presents sample Dynamo scripts prepared to demonstrate possibilities of the Evo package as well as the way in which the scripts may be built. All the samples are \*.dyn files and auxiliary custom nodes (\*.dyf files), saved in Dynamo, v. 1.3.3.

Samples are available in the package directory. For **Dynamo Core**:

```
%APPDATA%\Dynamo\Dynamo Core\1.3\packages\Evo\extra
or
%APPDATA%\Dynamo\Dynamo Core\2.0\packages\Evo\extra
```

For **Dynamo Revit**:

```
%APPDATA%\Dynamo\Dynamo Revit\1.3\packages\Evo\extra
or
%APPDATA%\Dynamo\Dynamo Revit\2.0\packages\Evo\extra
```

#### 3.1. Evo. DiagonalProblem sample

##### Problem Description

The purpose of this sample is to find start and end points of the longest line that can fit in a rectangle of given height and width. Such the trivial problem has a natural solution in the form of a diagonal of the rectangle, however the genetic algorithm itself has no idea about the geometrical background of the problem.

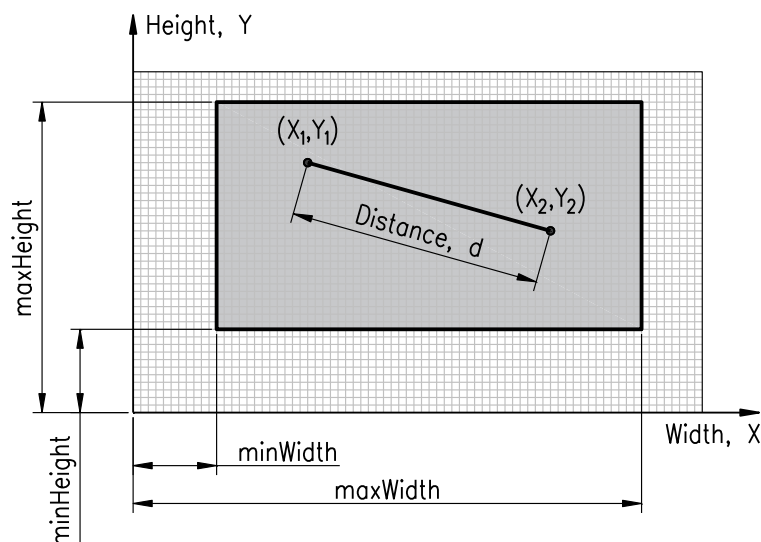


Fig. 12. Evo. DiagonalProblem. Visualization of the optimization task.



The idea is to generate random solution candidates. Each candidate consists of four numbers:  $X_1$  and  $Y_1$  for the start point and  $X_2$  and  $Y_2$  for the end point. For each set a fitness function value will be evaluated. The fitness function is Euclidean distance function that should be maximized. An exemplary solution candidate and its boundaries are presented in the Fig. 12.

The fitness function – Euclidean distance:

$$d = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

Constraints:

$$\text{minWidth} \leq X_1 \leq \text{maxWidth}, \text{minHeight} \leq Y_1 \leq \text{maxHeight}$$

$$\text{minWidth} \leq X_2 \leq \text{maxWidth}, \text{minHeight} \leq Y_2 \leq \text{maxHeight}$$

The genetic algorithm will try to find the solution by operating on the sets and function results only, manipulating them iteratively. In fact, it has no even access to the fitness function body. Instead, it utilizes only raw results the function returns for provided solution candidates. In the example, minimum width and height are set as 0.0, maximum width is 900.0 and maximum height is 600.0.

The optimization task is a four-variable problem. At the beginning of the script, a floating point chromosome is created using **Chromosomes.CreateFloatigPointChromosome** node. Its second variant is used, requiring following inputs: list of minimum values per each variable, list of maximum values per each variable and list of precisions (numbers of fraction digits) per each variable. All the three lists consists of four elements – one element per each variable.

Then a population of 50 solution candidates is generated. The population is passed to the **Basic.CreateGeneticAlgorithm** node. Default operators are used: Elite Selection, Uniform Crossover, Flip Bit Mutation, termination after 100 stagnant generations, selection size equal to the half of the population size, crossover and mutation probabilities: 0.75 and 0.10, respectively.

The Euclidean distance fitness function is included in the **DistanceBetweenPoints** custom DYF node. To inspect the node, click right mouse button on the node and, from the context menu, select: *Edit Custom Node*. Its structure is a nodal version of the formula shown above.

The total evolving time is about 2 seconds with 269 generations produced. The best solution is a combination of four values: 0, 0, 600 and 900 of the total length between points:  $d = 1081,7$  units. Note that there are two diagonals in the rectangle so the output numbers can have different order.

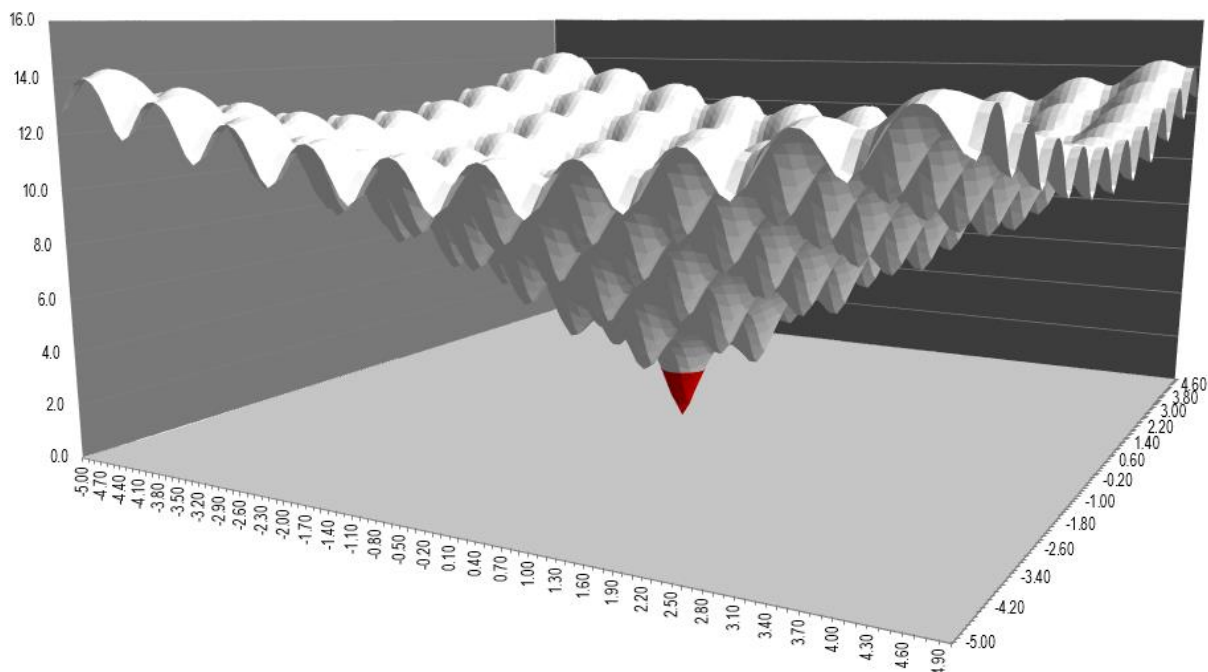
### 3.2. Evo. AckleyFunction sample

#### Problem Description


The Ackley function is a non-convex, two-variable function used as a performance test for optimization algorithms, originally proposed by David Ackley in 1987. The function can be expressed mathematically using the following formula:

$$f(x, y) = -20 \cdot e^{-\frac{1}{5} \sqrt{\frac{1}{2}(x^2 + y^2)}} - e^{\frac{1}{2}(\cos 2\pi \cdot x + \cos 2\pi \cdot y)} + e + 20$$

The function is characterized by numerous local optima and only one global minimum:  $f(0, 0) = 0$ . 3D plot for the Ackley function is presented below:



*Fig. 13. Evo. AckleyFunction. Visualization of the optimization task.*

The optimization task is to find the function global minimum using the visual programming environment. The chromosome pattern of the problem is a bit simpler than the one used in the sample 3.1. as two variables are considered, instead of four. Other most common test functions are listed [here](#) .

For better performance, a few modifications were introduced to the **AckleyFunction** custom node. First of all, since the implemented algorithms internally look for global maxima of defined functions, results of the formula above is multiplied by -1 right before passing it to the output. Moreover, the function originally operates on positive or negative float point variables. It should be noted that binary representation of negative numbers require 64 bits which is the maximum available length of the binary chromosome and significant numbers of its chars are long swaths of 0's or 1's. Also, the more fraction

digits required, the longer binary representation. Such the chromosomes may reduce the convergence rate. Therefore, the whole function plot has been moved towards the positive quadrant of the X-Y coordinate system. The modified formula has the following form:

$$f(x, y) = 20 \cdot e^{-\frac{1}{5} \sqrt{\frac{1}{2}[(x-a)^2 + (y-b)^2]}} + e^{\frac{1}{2}[\cos 2\pi \cdot (x-a) + \cos 2\pi \cdot (y-b)]} - e - 20$$

It was assumed  $a = b = 100$ . The new global optimum can be expected as  $f(100, 100) = 0$ . From now on, boundaries of the variables cover positive numbers only:  $0 \leq x \leq 200$  and  $0 \leq y \leq 200$ , reducing required length of encoded chromosomes. Required precision of variables was set as 1 for each variable.

At the beginning of the script, a floating point chromosome is created using **Chromosomes.CreateFloatigPointChromosome** node. Its second variant is used, requiring following inputs: list of minimum values per each variable, list of maximum values per each variable and list of precisions (numbers of fraction digits) per each variable. All the three lists consists of two elements – one element for  $x$  variable and one for  $y$  variable.

Then a population of 100 solution candidates is generated. The population is passed to the **Basic.CreateGeneticAlgorithm** node. Following operators are used: Tournament Selection, Voting Recombination Crossover, Flip Bit Mutation (by default), termination after 100 stagnant generations (by default), selection size equal to population size, crossover and mutation probabilities: 0.75 and 0.01, respectively.

The Ackley function is included in the **AckleyFunction** custom DYF node. To inspect the node, click right mouse button on the node and, from the context menu, select: *Edit Custom Node*. Its structure is a nodal version of the modified formula shown above.

The total evolving time is about 8 seconds, producing 114 generations. The best solution varies around the point  $x = 100.0$  and  $y = 100.0$ . In the example, total evolving time is inspected, together with total number of created generations, best chromosomes in progressive generations, fitness values associated with them and, finally, the best chromosome in the most recent population. Note that the function converges relatively quickly, producing small amount of progressive generation, that has an impact on the solution quality. Using a normalized fitness function, returning positive values from a range between 0.0 and 1.0 may be considered.

BestChromosome			
generation	>	bestChromosome	
fitnessFunction	>	fitness	
List			
0 List			
		99.9	
		98.8	
		-5.24985452170737	
@L3 @L2 @L1 {3}			

### 3.3. Evo. TravellingSalesmanProblem sample

#### Problem Description

The Travelling Salesman Problem (TSP) is a classical NP-hard combinatorial problem. Its input is a list of cities, in our case expressed in the form of points list (Fig. 14). The optimization task is to find the closest path between the points with no repetitions. The problem is visualized by a salesman who travels between the cities but during the route he cannot enter the same city twice and his path should form a closed curve.

The presented sample utilizes operations on graphical Dynamo objects. The first step in the solution search is formulating an adequate type of a chromosome pattern. In case of such the combinatorial problems, the dedicated chromosome can be created using **Chromosome.CreateOrderedChromosome**. The only input of the node is number of unique indices the chromosome should store.

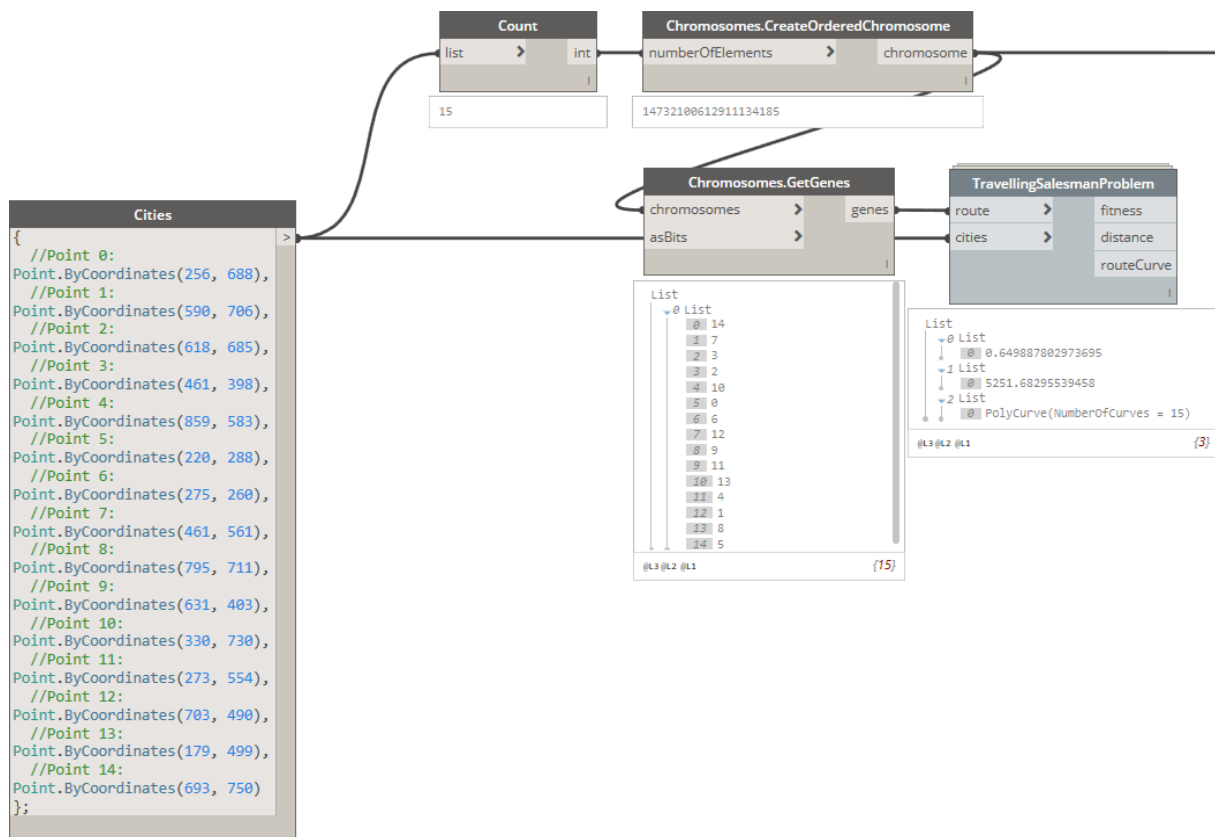
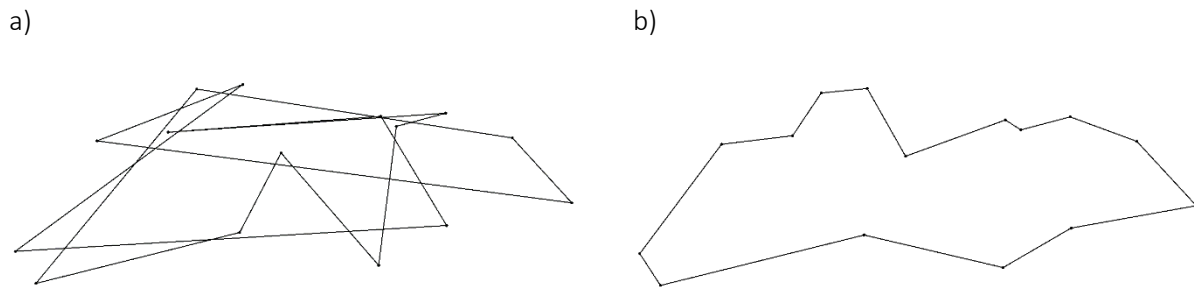


Fig. 14. Definition of an ordered chromosome, genes extracted out of the chromosome and the route calculated.

The **TravellingSalesmanProblem** is a custom DYF node responsible for calculating normalized fitness function value, real travel distance between the cities in turn and visualizing the route in the form of a polycurve. The travel distance is calculated by a **Curve.Length** node that gets the polycurve as an input, created by a **PolyCurve.ByPoints** node. All the operation can be checked within the DYF node.

Then a population of 100 random routes is generated. The population is passed to the **Basic.Create GeneticAlgorithm** node. Following operators are used: Roulette Wheel Selection, Ordered Crossover, Reverse Sequence Mutation, termination after 100 stagnant generations (by default), selection size equal to population size and default probabilities for both crossover and mutation: 0.75 and 0.05, respectively. Please keep in mind that some of the crossover methods, as well as some mutation methods, are designed especially for combinatorial types of optimization problems. The other ones operate on floating point chromosomes encoded to the binary strings (non-combinatorial problems).

The total evolving time is 15 to 30 seconds. Iterations were terminated after generation no. 201. The figure below shows the initial proposed route obtained by a random sequence between the cities. The total travel distance for the proposal is 5118.2 units. After search process the best route of total distance 2096.4 units was obtained, presented below to the right:



*Fig. 15. One of the initial route proposal (left) and the best solution obtained by the algorithm (right).*

## Scientific publications

Hussain, A., Muhammad, Y. S., Sajid, M. N., Hussian, I., Shoukry, A. M., Gani, S.: ***Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator***. Computation Intelligence and Neuroscience, vol. 2017, August 2017, pp. 1 – 7.

Konak, A., Coit, D. W., Smith, A. E.: ***Multi-Objective Optimization Using Genetic Algorithms: A Tutorial***. Reliability Engineering & System Safety, vol. 91, issue 9, September 2006, pp. 992 – 1007.

Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., Dizdarevic, S.: ***Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators***. Artificial Intelligence Review, vol. 13, issue 2, April 1999, pp. 129 – 170.

Umbarkar, A. J., Sheth, P. D.: ***Crossover Operators in Genetic Algorithms: A Review***. ICTACT Journal on Soft Computing, vol. 6, issue 1, October 2015, pp. 1083 – 1092.