cnc Documentation

Release 1.0.0

Stefan Petrovich

May 15, 2019

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Requirements

This package depends on the following packages, which are automatically installed with the provided installer:

1. Python 3
2. numpy – used for numerical computations (the optimization itself)
3. bokeh – used for visualization
4. tqdm – used for displaying a progress bar while optimizing

1.2 How To Install

This package is distributed using *setuptools*, and can be installed using the *setup.py* file provided with the package. The package is installed into the currently active Python environment using the following command:

```
>>> python setup.py install
```

CHAPTER

TWO

IMPLEMENTATION DETAILS

The problem has two requirements. One is that a certain line group order needs to be respected, and the other one is that lines can be cut either way.

2.1 How Group Order Is Preserved

The line order that needs to be respected is the following one:

1. REF
2. SCRIBE_LINE (non 2 recipe)
3. BUSBAR_LINE
4. EDGEDEL_LINE
5. SCRIBE_LINE2

In order for this grouping to be respected, the lines are grouped together after the .code file is parsed, and when the population matrix is being initialized, line groups are placed into the matrix left to right, column-wise, respecting the specified group order, using an ordered dictionary (standard in Python 3.7, but implemented as a special *OrderedDict* class in earlier versions).

While initializing the population matrix, pointers to parts of the population matrix are being constructed (so called numpy array views). So, one can access and manipulate every group of the population individually, without affecting the group ordering that was specified during the initialization of the population matrix. Hence, in the *crossover* and *mutation* methods, only these groups are used to performs these actions, so there's only inner-group mixing of genetic material (lines can't get out of their respective groups).

On the other hand, the path cost (and fitness) gets calculated "globally", that is, using the whole population matrix, not individually for every group. This ensures that individuals with best inner-group line ordering are favored.

2.2 How It's Optimized

There are two steps to the optimization. The first step is to find the best possible line order, taking into account that the lines can be oriented either way. This is done using the genetic algorithm with a heuristic instead of a precise fitness function. The heuristic tries to estimate the lowest possible path cost, if all the lines can be simultaneously oriented both ways, which means, that the heuristic gives better scores to line orders which can *potentially* have a very low path cost, if the right line orientation is found.

The second step is to find the best orientation for every line, for the line order that was determined by the genetic algorithm using the heuristic. This is done using the *hill-climbing* algorithm. While performing this part of the optimization, the real path cost is used instead of the heuristic.

CHAPTER

THREE

USAGE

3.1 CNCOptimizer

The *CNCOptimizer* class is the only class needed in order to run the optimization. An instance of the class gets initialized with the path to the .code file that needs to be processed, and a flag that tells the optimizer whether to ignore recipes or not. Two optinal arguments are available, that can be used to fine tune the scale of the optimization. The arguments are *time_factor*, which increases the number of iterations and the population size of the optimization, and *num_threads*, which controls the number of parallel optimizations that are run. The optimizations all have a different random seed, so they all find a slightly different solution, and the best one is taken as the result. The altorithm also automatically scales with problem difficulty (the number of lines), and the ammount of scaling is also controlled with the *time_factor* argument.

Visualization now takes longer than optimization. After instantiating an *CNCOptimizer* object, the *.optimize* method needs to be called, after which the *.save* method is used to save the result of the optimization to a .code file.

A visualization can be generated, if needed, using the *.visualize* method, which generates a *visualization.html* file, that can be viewed using any browser.

3.2 Full Example

A full example, which is also provided with this package, in the *run.py* file, is given below:

```
from cnc.optimization import CNCOptimizer

# Path to input file
path_file = './path_files/p3_stpl001.code'

# Scaling factor which determines the size of the optimization (length and
# scale) and hence the time as well
timing_factor = 1

# Number of threads to run for the optimization (default is 4)
num_threads = 4

# Generate optimization object
opt = CNCOptimizer(path_file, timing_factor, num_threads, recipe_grouping=True)

# Run the optimization
opt.optimize()

# Write the optimization to a file
opt.save('result')
```

(continues on next page)

(continued from previous page)

```
# Generate visualization file
opt.visualize()
```

CHAPTER

FOUR

REFERENCE

class cnc.optimization.**CNCOptimizer**(*file_path*, *time_factor=1*, *num_threads=4*, *recipe_grouping=True*)

Solves an instance of the travelling salesman problem, for the CNC machine.

Finds the shortest cutting tool travel path (SCTTP) using the genetic algorithm optimization method, and the best orientation for the lines using the hill-climbing algorithm.

Parameters

file_path [string] Path to the file containing the lines that need to be ordered.

time_factor [float] Scaling factor for the optimization, which increases the population and the number of generations of the algorithm.

num_threads [int] Number of parallel runs of the optimization. The best one is returned as the result.

recipe_grouping [bool] Determines whether the recipe numbers should be ignored or not.

Attributes

lines [list] Contains all of the lines, as Line objects, after the input file is parsed.

recipe_grouping [bool] Determines whether the recipe numbers should be ignored or not.

groups [OrderedDict] Key-value pairs where the key is the group name and the value is all the Line objects which belong to that group.

group_sizes [dict] Key-values pairs where the key is the group name and the value is the number of Line objects in that group.

num_genes [int] Total number of lines, which determines the width of the population matrix (the number of genes per individual).

time_factor [float] Scaling factor for the optimization, which increases the population and the number of generations of the algorithm.

pop_size_scaler [int] Scaling factor which is internal to the optimizer. Controls how the population size scales with problem difficulty.

num_generations_scaler [int] Scaling factor which is internal to the optimizer. Controls how the number of generations scales with problem difficulty.

population [numpy array] Contains all of the individuals in the optimization. When it gets constructed, it respects the specified group order (see *group_lines* and *generate_initial_population* method method).

sub_pops [dict of numpy arrays] Contains numpy array views of the population, for all the groups.

next_sub_pops: dict of numpy arrays Contains numpy array views of the next generation, for all the groups.

bi_directional_scaler [int] Scaling factor which is internal to the optimizer. Controls how the number of iterations for hill-climbing scales with problem difficulty.

num_threads [int] Number of parallel runs of the optimization. The best one is returned as the result.

pop_size [int] Number of individuals in the population, or the number of rows in the population matrix. Calculated based on problem difficulty and scaling factors.

maximum_distance [float] The maximum distance between any two lines, in any orientation. Used when calculating the fitness of the population.

prob_mut [float] Probability of mutation for any individual in the population.

num_mut [int] Number of mutations in the population, calculated based on *prob_mut*.

num_generations [int] Number of iterations for which to run the genetic algorithm. Calculated based on problem difficulty and scaling factors.

best_result [dict] Dictionary containing the solution and non-cutting path cost of the best individual in the optimization.

initial_result [dict] Dictionary containing the solution and non-cutting path cost of one individual at the start of the optimization. Used for comparison in the visualization.

path_cost [numpy array] Vector containing the path cost of every individual in the population.

fitness [numpy array] Vector containing the fitness of every individual in the population. It's calculated by subtracting the *path_cost* from the maximum path cost possible.

ONES [numpy array] Matrix of ones, used for constructing a matrix of path costs, when doing crossover.

Methods

bi_directional(best_list, progress_bar_position)	Uses hill-climbing to find the best orientation for every line, so that it minimizes the path cost.
crossover()	Generates the next generation using crossover.
evaluate_generation()	Evaluates the path cost and hence fitness of the whole generation, using a heuristic.
generate_distance_matrix()	Generates matrix of Euclidian distances between every two nodes.
generate_initial_population()	Generates initial population, while also implementing correct group order, and constructing views that point to slices of the population that represent every group.
generate_lines_from_file(file_path)	Parses the input file generates Line objects for every line, and stores them in a list.
get_initial()	Returns order of Line object obtained at the beginning of optimization.
get_result()	Returns the correctly flipped, optimized order of Line objects.
group_lines()	Groups Line objects, and stores the group in an ordered dictionary.
mutation()	Performs mutation by swapping two genes around.

Continued on next page

Table 1 – continued from previous page

opt_thread(best_list, initial_list, ...)	Gets launched as a child process by the main process and performs one optimization, with a newly generated random seed.
optimize()	Runs the specified number of threads, each doing a complete path and direction optimization with a different random seed.
save(file_name)	Saves the results of the optimization to a file.
visualize()	Visualizes the result of the optimization, using the Visualizer class.

bi_directional(*best_list*, *progress_bar_position*)

Uses hill-climbing to find the best orientation for every line, so that it minimizes the path cost.

Parameters

best_list [process manager list] List that is shared between all threads, to which the result of the optimization gets appended to.

progress_bar_position [int] Determines the row in which the tqdm progress bar gets drawn. Also determines part of the thread name.

crossover()

Generates the next generation using crossover. The group order is kept intact while doing this.

Takes two individuals at a time, using a roulette game based on fitness, and crosses them using the Order 1 Crossover method. There are some clever numpy tricks used in this method (for performance reasons), which are not very verbose/intuitive, so I included a lot of commentary above these lines.

evaluate_generation()

Evaluates the path cost and hence fitness of the whole generation, using a heuristic.

The path cost is calculated using the Euclidean distance between every two successive lines in an individual. The heuristic that is used tells the evaluation to treat every so as if it can be oriented both ways simultaneously, so we're actually trying to find the lowest "possible" path cost (not the real path cost). We later use hill-climbing to find the best orientation for every line so that we get the real lowest path cost.

generate_distance_matrix()

Generates matrix of Euclidian distances between every two nodes.

The distances are generated for every pair of lines, in any orientation, meaning that for every two lines there are four distances. So the matrix is of 2Nx2N dimensions, where N is the number of lines.

generate_initial_population()

Generates initial population, while also implementing correct group order, and constructing views that point to slices of the population that represent every group.

generate_lines_from_file(*file_path*)

Parses the input file generates Line objects for every line, and stores them in a list.

Parameters

file_path [str] Path to the file containing the lines that need to be optimized.

get_initial()

Returns order of Line object obtained at the beginning of optimization.

Returns

out [list of Line] Lines ordered in a random fashion, from an individual at the beginning of optimization.

get_result()
Returns the correctly flipped, optimized order of Line objects.

Returns

out [list of Line] Correctly flipped, optimized order of Line objects

——-

group_lines()
Groups Line objects, and stores the group in an ordered dictionary.

The lines are stored in an ordered dictionary, since we need the groups to be in the correct order, according to the following requirements:

We have to respect the following order: 1) REF 2) SCRIBE_LINE (non 2 recipe) 3) BUSBAR_LINE 4) EDGEDEL_LINE 5) SCRIBE_LINE2

We need this order when iterating over the dict, while generating the initial population (see *generate_initial_population* method).

mutation()
Performs mutation by swapping two genes around. The group order is kept intact while doing this.

opt_thread(*best_list*, *initial_list*, *progress_bar_position*)
Gets launched as a child process by the main process and performs one optimization, with a newly generated random seed.

Parameters

best_list [process manager list] List that is shared between all threads, to which the result of the optimization gets appended to. Result gets appended after orientation optimization.

initial_list [process manager list] List that is shared between all threads, to which an individual from the start of the optimization gets appended to (used for visualization purposes).

progress_bar_position [int] Determines the row in which the tqdm progress bar gets drawn. Also determines part of the thread name.

optimize()
Runs the specified number of threads, each doing a complete path and direction optimization with a different random seed. Stores the best result of all the optimization runs.

save(*file_name*)
Saves the results of the optimization to a file. Adds .code file extension if it's not specified.

Parameters

file_name [str] Filename of the file which will contain the optimized result.

visualize()
Visualizes the result of the optimization, using the Visualizer class.

class cnc.optimization.**Line**(*line_type*, *starting_point*, *endpoint*, *recipe*, *line_id*)
Line which represents where the CNC head will perform cutting.

Parameters

line_type [str] Name of the line type, contained in the 1st column of the .code file.

starting_point [np.array] Numpy array of two coordinates, X1 and Y1, representing the starting point of cutting.

endpoint [np.array] Numpy array of two coordinates, X2 and Y2, representing the endpoint of cutting.

recipe [str] Recipe number, last column of .code file.

line_id [int] Unique line ID, used when constructing the initial population, so as to order the lines correctly (to respect the group order).

flip [book] Determines whether the starting and endpoints need to be flipped or not.

Methods

flip_line()	Flips the starting end endpoints.
get_endpoint()	Returns the endpoint of a line.
get_line_id()	Returns the ID of the line.
get_line_type()	Returns the type of line.
get_recipe()	Returns the recipe of a line.
get_starting_point()	Returns the starting point of a line.
get_thickness()	Returns thickness of EDGEDEL_LINE line type.
set_thickness(thickness)	Set the line thickness, which is only specified for EDGEDEL_LINE line types.

flip_line()
Flips the starting end endpoints.

get_endpoint()
Returns the endpoint of a line.

Returns

out [np.array] Numpy array of two coordinates, X2 and Y2, representing the endpoint of cutting. Returns the starting point if lines are flipped.

get_line_id()
Returns the ID of the line.

Returns

line_id [int] Number representing the ID of the line, which was given to it while parsing the input file.

get_line_type()
Returns the type of line.

Returns

line_type [str] Name of line type.

get_recipe()
Returns the recipe of a line.

Returns

recipe [str] Recipe number.

get_starting_point()
Returns the starting point of a line.

Returns

out [np.array] Numpy array of two coordinates, X1 and Y1, representing the starting point of cutting. Returns endpoint if lines are flipped.

get_thickness()
Returns thickness of EDGEDEL_LINE line type.

Returns

thickness [str] Number representing the thickness of the line.

set_thickness(*thickness*)
Set the line thickness, which is only specified for EDGEDEL_LINE line types.

Parameters

thickness [str] Number representing the thinkess of the line. Not used for calculations, only when writing to new .code file.

class cnc.visualization.**Visualizer**(*result*, *initial*)
Visualizes the result of the optimization for the CNC shortest cutting tool travel path, using bokeh.

Parameters

result [list of Line] List of Line objects, sorted in the order that represents the result of the optimization.

initial [list of Line] List of line objects, obtained at the beginning of the optimization, used for comaprison.

Methods

generate_tool_path(data, step_size)	Generates the whole trajectory of the cutting tool, with a step of *step_size*.
populate_plot(plot, data)	Adds data to the plot, like starting and endpoints of cutting, lines representing the cutting path and lines representnig the non-cutting path.
split_line(start, end, increment)	Function that generates a number of points between two points.
visualize()	Generates a plot using bokeh, which displays the initial trajectory and the optimized trajectory of the cutting tool.

generate_tool_path(*data*, *step_size*)
Generates the whole trajectory of the cutting tool, with a step of *step_size*.

Parameters

data [list of Line] List of line objects, from which the trajectory needs to be generated.

step_size [int] The Euclidian distance between two steps of the trajectory.

Returns

out [touple of np.arrays] Touple where the 1st and 2nd element represents the X and Y coordinates of the whole cutting tool trajectory, respectively.

populate_plot(*plot*, *data*)
Adds data to the plot, like starting and endpoints of cutting, lines representing the cutting path and lines representnig the non-cutting path.

Parameters

plot [bokeh figure object] Figure object on which the content of the *data* object will be displayed.

data [list of Line] List of line objects, which have to be drawn on the figure.

Returns

plot [bokeh figure object] Figure populated with data from the *data* object.

split_line(*start*, *end*, *increment*)

Function that generates a number of points between two points.

Generates two vectors, which represent the X and Y coordinates between points *start* and *end*.

Parameters

start [np.array] Numpy array containing X and Y of the starting point.

end [np.array] Numpy array containing X and Y of the endpoint.

increment [int] Euclidian distance between two successive entries in the *start* and *end* points.

Returns

out [touple of np.arrays] Touple where the 1st element is the X coordinates and the 2nd element is the Y coordinates of the line.

visualize()

Generates a plot using bokeh, which displays the initial trajectory and the optimized trajectory of the cutting tool.

PYTHON MODULE INDEX

C

INDEX