Documentation of the FuzzyCSP Library

The library is an encompassing pipeline for modeling Fuzzy constraint satisfaction problems, solving them with some common algorithms and approaches used in literature, and providing a way to create custom problems based on various input metrics. It is implemented in Python, and assumes the presence of library NumPy (which comes by default on most systems with Python). For the genetic algorithm approach, we introduce the use of one more library, PyEvolve, which can be found at http://pyevolve.sourceforge.net/0_6rc1/intro.html# .

The library provides these main files inside src folder:

`fuzzy_cs_problem.py`

`fuzzy_example_problem.py`

`fuzzy_cs_solution.py`

`university_course_selection_ga.py`

`n_queen_ga.py`

fuzzy_cs_problem.py

`fuzzy_cs_problem.py` has a class `FuzzyCSProblem`.

It takes the parameters “variables”, “domains”, and “constraints” to create a fuzzy problem instance.

The code demonstrating the data structures use of `FuzzyCSProblem` are:

```
variables = ['f', 't', 's']
domains = [('S', 'C'),('D', 'B', 'G'),('L', 'W')]
constraints = [
           {('S','D', None):1.0,
            ('S','B',None):0.4,
            ('S','G',None):0.2,
            ('C','G',None):0.8,
            ('C','B',None):0.5},
           {('S', None, 'L'):1.0,
            ('S', None, 'W'):0.7,
            ('C', None, 'W'):1.0,
            ('C', None, 'L'):0.1},
           {(None, 'D', 'W'):1.0,
            (None, 'D', 'L'):0.7,
            (None, 'B', 'W'):1.0,
```

```
            (None, 'B', 'L'):0.4,
            (None, 'G', 'L'):1.0,
            (None, 'G', 'W'):0.6}
            ]
problem = FuzzyCSProblem(variables,domains, constraints)
```

As clear from the code, “variables” is an array of variables, “domains” is an array of tuples each of which is the domain of corresponding variable (of same index) in the “variables” array.

“constraints” is an array of dictionaries, each dictionary representing a constraint. A constraint has a set of fuzzy satisfaction degrees to a fixed variables input. For example, in the example above, the first constraint dictionary enlists the fuzzy satisfaction values for different values of the variables ‘f’ and ‘t’.

In the “constraints” data structure, if a constraint between any of the variables is not given, i.e. one of the dictionaries is lacking, we assume that those variables are unconstrained. In the example above, all possible binary constraints for the variables (3 choose 2 = 3) are mentioned, so all pairs of variables are constrained in some way. Within an individual constraint (=a dictionary), if a fuzzy assignment for one of the value combinations is lacking, we assume that those values don’t go together, i.e. satisfaction degree for those values is 0. In the example above, first constraint lacks a candidate assignment `('C', 'D', None)`. Therefore, we assume that the value for that assignment it is 0.0.

Public functions:

- `get_variables()`: returns all variables
- `get_domain(variable)`: returns domain of a variable as a tuple of values that the variable can take
- `get_domains()`: returns all domains of all the variables in matching order of the variables
- `get_constraints()`: returns all the constraints of the problem (same data type as described above)

fuzzy_example_problem.py

This file contains a class FuzzyExampleProblem that is an interface of getting various example problems to be used for the solution. The main capability of this interface is that it can generate problems according to some problem parameters supplied to it.

The instantiation of the FuzzyExampleProblem takes the following parameters, so that it can generate a problem with these problem parameters:

- `num_variables` : number of variables desired for the problem to be generated (default = 3)
- `domain_size_per_variable`: number of values per variable (default =2)

- `num_vars_per_constraint` : number of variables that each constraint takes (default = 2)
- `tightness` : proportion of assignment of variables disallowed by each constraint, i.e. if there are n possible combination of variable assignments for a constraint, n*tightness variables don't satisfy the constraint, or have the satisfaction degree of 0 (default =0.25)
- `connectivity` : proportion of variables which are constrained. If 2 variables have the satisfaction degree of 1 for all their values, they are not constrained, i.e. not connected. (default = .75) #.25 ,.5, .75
- `range_sat_values` : range of values for degree of satisfaction for each constraint. The values are uniformly divided in the given range.(default= [0.5, 1.0])

Following is an example of instantiation:

```
num_variables = 3
domain_size_per_variable = 2
num_vars_per_constraint = 2
tightness = 0.25
connectivity = .75
range_sat_values = [0.5, 1.0]

example_problem_instance = FuzzyExampleProblem(num_variables,
domain_size_per_variable, num_vars_per_constraint, tightness,
connectivity, range_sat_values)
```

Public functions:

- `generate_example_problem()`: This is the most important public function. It returns an instance of FuzzyCSProblem from the parameters provided.
- `get_robot_dressing_problem()`: This returns an example problem of robot dressing problem. See method definition for variables, values, and constraints it takes.
- `get_heuristic_backtracking_problem()` : returns a variation of robot dressing problem that requires the heuristic search algorithm to backtrack
- `get_temporal_constraint_problem()` : returns a FuzzyCSProblem that models a temporal constraint problem
- `get_lunch_swim_problem()` : returns the example given by Francesca Rossi during her lecture of preference modeling

`fuzzy_cs_solution.py`

`fuzzy_cs_solution.py` contains two classes `FuzzyCSSolution`, where most of the code is, and `FuzzyBenchmarkMetrics`, where the performance metrics of the solution are stored if "`do_benchmark`" variable of the `FuzzyCSSolution` instance is set to `True`.

The primary class `FuzzyCSSolution` takes the input parameters "`problem`", "`joint_constraint_type`", "`upper_bound_type`", and "`do_benchmark`".

`problem` - `FuzzyCSProblem` instance:

- `joint_constraint_type` : approach to calculate the joint satisfaction degree of an instance. It can take the string values "productive", "average", or "min" depending on what type of joint is desired. Default value is "productive".
- `upper_bound_type` : required for branch and bound algorithm for a type of upper bound. It takes a string, either "appropriateness" or "partial_joint_sat". Default value is "appropriateness"

This is an example code of instantiating a `FuzzyCSSolution` instance, given an instance "problem" of FuzzyCSProblem:

```
problem = FuzzyExampleProblem().get_robot_dressing_problem()
joint_sat_type = "productive"
upper_bound_type = "appropriateness"

solution = FuzzyCSSolution(problem, joint_sat_type, upper_bound_type)
```

Public functions:

These are the main public functions (omitting the simple getter/setter functions to get/set instance variables):

- `get_joint_satisfaction_degree(instantiation)` - `instantiation`: a solution instance/instantiation is a tuple of length equal to the number of variables, each ith element of tuple equal to one of the values in the domain of ith variable. In short, it's an assignment of all variables
  returns a float representing degree of joint satisfaction of the solution instance by all the constraints in the problem. The degree is calculated based on "joint_constraint_type" provided to the instance.
- `get_heuristic_solution()` : returns a solution instance by running heuristic search algorithm
- `get_branch_and_bound_solution()` : returns a solution instance given by the branch and bound algorithm
- `get_alpha_solutions_backtracking(alpha)` : returns so-called "alpha solutions", i.e. all solution instances having joint satisfaction degree >= alpha, with backtracking algorithm. It is a so-called "generator function" in python, which means it gives each solution instance one by one as it finds, so to get a list of all solution instances, we need to do: `list(get_alpha_solutions_backtracking(alpha))`
- `get_alpha_solutions_branch_n_bound(alpha)`: Same as above, but with branch and bound algorithm
- `get_all_feasible_solutions_backtracking()` : returns all feasible solution instances (= joint satisfaction degree >0) with backtracking algorithm

- `get_a_feasible_solution_backtracking()` : returns the first feasible solution found by the backtracking algorithm
- `get_an_alpha_solution_backtracking(alpha)` : returns the first alpha solution found by the backtracking algorithm
- `get_an_alpha_solution_branch_n_bound(alpha)` : returns the first alpha solution found by the branch and bound algorithm

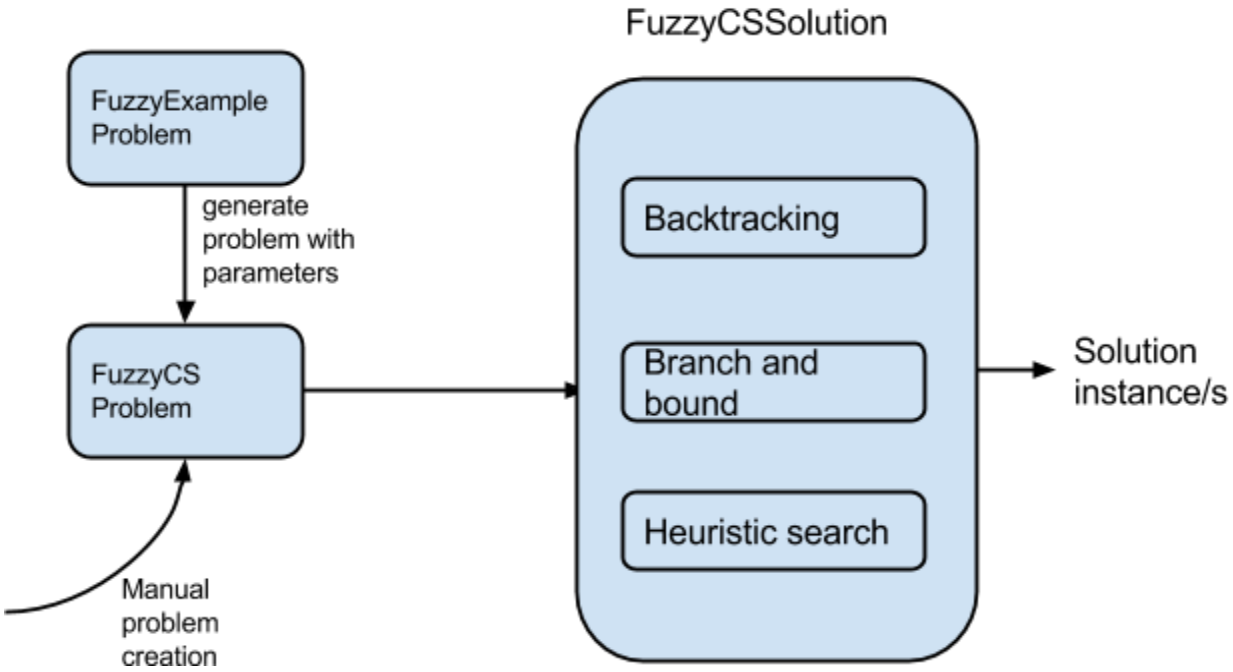

Fig: Simple system diagram of the FuzzyCSP library

Things to note when using the library:

1. Multiple variables cannot have values with same name. If they have same name, please attach a suffix to make them distinct. Eg. if variables ‘a’ = {‘W’, ‘M’, ‘H’} and ‘b’ = {‘W’,’M’,’H’}, please change the problem to: ‘a’ = {‘W_a’, ‘M_a’, ‘H_a’}, ‘b’={‘W_b’, ‘M_b’, ‘H_b’}. This is because while constructing the search tree, if there are duplicated domain names, there will be duplicate keys in the dictionary, which we have not handled.

2. We have only tested with strings for variable and value names. Feel free to use numerical or other characters, but no guarantees.

Documentation for Genetic Algorithms:

The library provides genetic algorithms for solving two specific problems.
It is implemented in Python, and assumes the use of PyEvolve, which can be found at http://pyevolve.sourceforge.net/0_6rc1/intro.html# .
Each python file includes the representation for the problem (as an evaluation function), the main algorithm, and the code for providing the algorithm performance (number of generations, runtime, etc.)

This library provides these main files inside the **test** folder:

`university_course_selection_ga.py`

`n_queen_ga.py`

university_course_selection_ga.py

This file contains a genetic algorithm for solving the university course selection problem from the tutorial. It has two main parts/functions:

- `eval_func(ind)`: This evaluation function is called for each individual (ind) to evaluate its fitness. Each individual is represented as a list with length 3.
- `run_main()`: runs the main algorithm for finding the solution to the problem.

For a basic GA, `run_main()` includes the following (common) functions:

- `genome.setParams(parameters)`: Sets the parameters for the genome. In this case, we choose the parameters (`rangemin, rangemax, rounddecimal`), which define how the genes for each individual are encoded. Genes are encoded using numbers from the range `[rangemin, rangemax]`, and the numbers are rounded up to `rounddecimal` points.
- `genome.initializator.set(initializator_type)`: Sets/initializes the encoding for the chromosomes. In our case we use `Initializators.G1DListInitializatorInteger` for the integer encoding of the chromosomes, and `Initializators.G1DListInitializatorReal` for real encoding of the chromosomes.
- `genome.mutator.set(type_of_mutator)`: sets the type of mutator for the genome
- `genome.crossover.set(type_of_crossover)`: sets the type of crossover for the genome
- `genome.evaluator.set(function)`: sets the function used for evaluating the genome's fitness.
- `ga.setMinimax(minimaxType)`: sets the type of (minimax) operation on the evaluation function (eval_func). We can choose between maximizing the evaluation

function (`Consts.minimaxType["maximize"]`), or minimizing (`Consts.minimaxType["minimize"]`)
- `ga.setPopulationSize(number)`: sets population size for the genome
- `ga.setGenerations(number)`: sets the number of generations for the genome
- `ga.setMutationRate(rate)`: sets the mutation rate for the genome
- `ga.setCrossoverRate(rate)`: sets the crossover rate for the genome
- `ga.selector.set(type_of_selector)`: sets the survival selection type. If nothing specified, the default type is to replace the worst.
- `ga.evolve(frequency)`: sets the frequency of the evolution
- `ga.bestIndividual()`: returns the best individual, i.e, the best solution found for the problem.

By varying the parameters for these functions, we can obtain different solutions and/or different runtimes for the solutions.

n_queen_ga.py

This file contains a genetic algorithm for solving the n-queen problem from the tutorial. Its most important parts are:

- `queens_eval(genome)`: Evaluation function for the genome. We start with a score of BOARD_SIZE, and decrease the score for each collision (queen attacking another).
- `run_main()`: runs the main algorithm for finding the solution to the problem.

Some new functions appear in `run_main()` for this algorithm:

- `genome.setParams(bestrawscore, rounddecimal)`: In this case, one of the parameters chosen is bestrawscore, which will be useful when it comes to the termination criteria.
- `ga.terminationCriteria.set(criteria)`: sets the termination criteria for the algorithm. If it is not specified, the algorithm will stop once all the generations are run. In this case the criteria is set to `GSimpleGA.RawScoreCriteria`. The algorithm will stop once the genome achieves the score of bestrawscore.

Besides the parameters for the functions mentioned in the previous problem, to obtain different solutions and runtimes, we can also change the BOARD_SIZE and bestrawscore.