tvm.autotvm.MYGA

包含改进遗传算法的各类参数的枚举类及对应工具类，包括selection,crossover,mutation,aga,sa

包含配置类，用以设置遗传算法的各类参数

包含ga，用以进行遗传算法的迭代

tvm.autotvm.MYGA.selection

包含枚举类EnumSelection和工具类GASelection

EnumSelection

继承Enum，包含已配置选择算子种类，并可返回一随机选择算子

```
class EnumSelection(Enum):
    dup   = 0
    ecs   = 1
    etour = 2
    otos  = 3
    rcs   = 4
    rps   = 5
    rws   = 6
    sus   = 7
    tour  = 8
    urs   = 9
    crws  = 10
    @staticmethod
    def randomSelection():
        l = len(EnumSelection)
        idx = np.random.randint(l)
        selection = EnumSelection(idx)
        return selection
```

GASeletion

提供具体选择算子函数，包括dup，ecs等11种选择算子函数

如

```
@staticmethod
def dup(genes, scores, num=2):
    tmp_scores = np.copy(scores)[:, np.newaxis]
    p1 ,p2 = geatpy.dup(tmp_scores, num)
    p1 %= len(genes)
    p2 %= len(genes)
    return  p1, p2
```

tvm.autotvm.MYGA.crossover

包含枚举类EnumCrossover和工具类GACrossover

EnumCrossover

继承Enum，包含已配置交叉算子种类，并可返回一随机交叉算子

```
class EnumCrossover(Enum):
    spc   = 0 # single point cross
    tpc   = 1 # two point cross
    mpc   = 2 # multipoint cross
    ec    = 3 # even cross
    etpc = 4 # Even two-point cross
    dc   = 5 # Discrete cross
    ac   = 6 # Arithmetic cross
    hc   = 7 # Heuristic cross

    @staticmethod
    def randomCrossover():
        l = len(EnumCrossover)
        idx = np.random.randint(l)
        crossover = EnumCrossover(idx)
        return crossover
```

GACrossover

提供具体交叉算子函数，包括spc，tpc等8种交叉算子函数

如

```
@staticmethod
def spc(g1, g2, size):
    if size >= 1:
        point = np.random.randint(size)
        tmp_gene = g1[:point] + g2[point:]
        return tmp_gene
    return g1
```

tvm.autotvm.MYGA.mutation

包含枚举类EnumMutation和工具类GAMutation

EnumMutation

继承Enum，包含已配置变异算子种类，并可返回一随机变异算子

```
class EnumMutation(Enum):
    rm = 0 # random mutation
    cm = 1 # crossover mutation
    sm = 2 # swap mutation
    im = 3 # inversion mutation
    Rm = 4

    @staticmethod
    def randomMutation():
        l = len(EnumMutation)
        idx = np.random.randint(l)
        mutation = EnumMutation(idx)
        return mutation
```

GAMutation

提供具体变异算子函数，包括rm，cm等5种变异算子函数

如

```
@staticmethod
def rm(gene, dims, rate):
    for j, dim in enumerate(dims):
        if np.random.random() < rate:
            gene[j] = np.random.randint(dim)
            return gene
```

tvm.autotvm.MYGA.aga

增加自适应遗传算法，根据适应度动态调整个体交叉率和变异率，包含枚举类EnumAGA和工具类AGA

EnumAGA

继承Enum，包含已配置AGA算子种类，并可返回一随机AGA算子

```
class EnumAGA(Enum):
    GA   = 0 # normal Genetic Algorithm
    AGA  = 1 # Adaptive Genetic Algorithm
    LAGA = 2 # Linear Adaptive Genetic Algorithm
    CAGA = 3 # Cosine Adaptive Genetic Algorithm
    IAGA = 4 # Improved Adaptive Genetic Algorithm

    @staticmethod
    def randomAGA():
        l = len(EnumAGA)
        idx = np.random.randint(l)
        aga = EnumAGA(idx)
        return aga
```

AGA

提供具体AGA算子函数，包括GA，AGA等5种AGA算子函数

如

```
@staticmethod
def AGA(k1, k2, f_max, f_avg, f, target):
    if f < f_avg:
        return k2
    else:
        return k1 * (f_max - f) / (f_max - f_avg)
```

tvm.autotvm.MYGA.sa

增加模拟退火算法（SA），搭配GA使用，包含枚举类EnumAGA，具体算法见tvm.autotvm.MYGA.ga.

EnumSA

继承Enum，包含已配置SA算子种类，并可返回一随机SA算子

```
class EnumSA(Enum):
    GA     = 0
    GA_SA1 = 1
    GA_SA2 = 2

    @staticmethod
    def randomSA():
        l = len(EnumSA)
        idx = np.random.randint(l)
        sa = EnumSA(idx)
        return sa
```

tvm.autotvm.MYGA.config

包含GA各参数对应配置类，包括SelectionConfig，CrossoverConfig，MutationConfig，ExtraConfig，PopulationConfig，MultipopulationConfig

SelectionConfig

设置选择算子，其类型必须是EnumSelection或其name对应的str

CrossoverConfig

设置交叉算子，其类型必须是EnumCrossover或其name对应的str

设置交叉率及最大最小值

MutationConfig

设置变异算子，其类型必须是EnumMutation或其name对应的str

设置变异率及最大最小值

ExtraConfig

设置AGA，其类型必须是EnumAGA或其name对应的str

设置SA，其类型必须是EnumSA或其name对应的str

PopulationConfig

设置种群参数，包括种群规模，精英数量，SelectionConfig，CrossoverConfig，MutationConfig，ExtraConfig

输入为dict类型，例

```
    config = {
        "pop_size": 200,
        "elite_num": 3,
        "selection": {
            "op": "dup" # "dup" also can be EnumSelection.dup
        },
        "crossover": {
            "op": "tpc", # "tpc" also can be EnumCrossover.tpc
            "rate": 0.6
        },
        "mutation": {
            "op": "im", # "im" also can be EnumMutation.im
            "rate": 0.001
        }
    }
```

MultipopulationConfig

设置多种群参数，输入为dict类型，包含各种群对应参数，例

```
    config = {
        "config1" : {
        "pop_size": 100,
        "elite_num": 3,
        "selection": {
            "op": "dup"
        },
        "crossover": {
            "op": "tpc",
            "rate": 0.6
        },
        "mutation": {
            "op": "im2",
            "rate": 0.02
        }
    },
        "config2": {
            "pop_size": 200,
            "elite_num": 4,
            "selection": {
                "op": "rws"
            },
            "crossover": {
                "op": "spc",
                "rate": 0.6
            },
            "mutation": {
                "op": "rm",
                "rate": 0.001
            }
        },
    }
```

tvm.autotvm.MYGA.ga

用以进行遗传算法的具体计算，包括类Solution，Population，Multipopulation

Solution

种群个体，包括基因及适应度

```
class Solution:
    def __init__(self, gene, score=0.0):
        self.gene = gene
        self.score = score
```

Population

核心类，单种群GA，下面具体介绍其函数

Population.__init__

```
    def __init__(self, tunerConfig, popConfig : PopulationConfig):
        self.tunerConfig = tunerConfig # tuner config,don't need to care
        self.popConfig = popConfig # pop config,set selection op and other op
        self.space = len(self.tunerConfig.space) # get search space size
        self.popConfig.setSize(min(self.popConfig.getSize(), self.space)) # pop
size can't exceed the search space size

        self.solutions = [] # pop individuals
        self.elites = [] # pop elites

        self.genes = [] # individual's gene
        self.scores = [] # individual's score

        self.tmp_solution = []

        self.next_solution = []
        self.next_batch = []
        self.next_id = -1 # index of next individual
```

其中tunerConfig为搜索空间参数，tuner自动生成，无需关注

popConfig为种群配置类，包括该种群的各具体参数

Popolation.init

产生足够的随机

```
def init(self):
    # random initialization
    for _ in range(self.popConfig.getSize()):
        tmp_gene = point2knob(np.random.randint(self.space),
self.tunerConfig.dims)
        tmp_solution = Solution(tmp_gene)
        self.solutions.append(tmp_solution)
```

Population.update

```
    def update(self):
        # check whether to use SA or not
        if self.popConfig.getExtraConfig().getSA() == EnumSA.GA: # don't use SA
            self.genes = []
            self.scores = []
            for solution in self.solutions:
                if solution.score > 0:
                    self.genes.append(solution.gene)
                    self.scores.append(solution.score)
            self.crossover()
            self.mutation()
            self.next_id = -1
        else: # use SA
            self.T = 4 # The initial temperature of SA
            self.a = 0.5 # the decrease rate of SA
            self.Tmin = 1 # the lowest temperature of SA
            while self.T > self.Tmin: # SA begin
                self.genes = []
                self.scores = []
                for solution in self.solutions:
                    if solution.score > 0:
                        self.genes.append(solution.gene)
                        self.scores.append(solution.score)
                self.crossover()
                self.mutation()
                self.T *= self.a
                self.next_id = -1
```

Population.get_max_and_avg_and_min

返回种群适应度的最大值，平均值和最小值

```
def get_max_and_avg_and_min(self, solutions):
    f_max = 0
    f_min = solutions[0].score
    f_avg = 0
    sum = 0
    num = 0
    for solution in solutions:
        if solution.score > f_max:
            f_max = solution.score
        if solution.score < f_min:
            f_min = solution.score
        if solution.score != 0.0:
            sum += solution.score
            num += 1

    if num != 0:
        f_avg = sum / num

    return f_max , f_avg, f_min
```

Population.crossover

```
def crossover(self):
    self.tmp_solution = []

    dims = self.tunerConfig.dims
    size = self.popConfig.getSize()

    # get selection and crossover op
    selectionOP = self.popConfig.getSelectionConfig().getOP()
    crossoverOP = self.popConfig.getCrossoverConfig().getOP()

    # get the function corresponding to the selection and crossover op
    selectMethod = getattr(GASelection, str(selectionOP.name))
    crossoverMethod = getattr(GACrossover, str(crossoverOP.name))

    # get AGA
    aga = self.popConfig.getExtraConfig().getAGA()
    AGAMode = False
    toEvaluate = []
    if aga != EnumAGA.GA:
        f_max , f_avg, f_min= self.get_max_and_avg_and_min(self.solutions)
        if f_max != 0 :
            AGAMode = True

        AGAMethod = getattr(AGA, str(aga.name))

    # begin to crossover
    while len(self.tmp_solution) < size:
        if len(self.genes) >= 2:
            p1, p2 = selectMethod(self.genes, self.scores)
            s1, s2 = self.solutions[p1], self.solutions[p2]
            if s1.score < s1.score:
                s1, s2 = s2, s1
        else:
            s1 = s2 = self.solutions[0]
        if AGAMode:
            rate = AGAMethod(self.popConfig.getCrossoverConfig().min_rate,
                             self.popConfig.getCrossoverConfig().max_rate,
                             f_max, f_avg, s1.score, target = 0)
        else:
            rate = self.popConfig.getCrossoverConfig().getRate()
        if np.random.random() < rate:
            if crossoverOP == EnumCrossover.dc:
                if len(self.genes) >= 2:
                    tmp_gene = GACrossover.dc(self.genes,
len(self.tunerConfig.dims))
                else:
                    tmp_gene = GACrossover.dc([s1.gene, s2.gene],
len(self.tunerConfig.dims))
            elif crossoverOP == EnumCrossover.hc:
                tmp_gene = GACrossover.hc(s1.gene, s2.gene,
len(self.tunerConfig.dims), self.tunerConfig.dims, self.space)
            else:
                tmp_gene = crossoverMethod(s1.gene, s2.gene,
len(self.tunerConfig.dims))
            if self.popConfig.getExtraConfig().getAGA() != EnumAGA.GA or
self.popConfig.getExtraConfig().getSA() != EnumSA.GA:
                score = (s1.score + s2.score) / 2
            else:
                score = 0.0
            self.tmp_solution.append(Solution(tmp_gene, score))
```

Population.mutation

```
def mutation(self):
    next_solution = []

    aga = self.popConfig.getExtraConfig().getAGA()
    f_max, f_avg, f_min = self.get_max_and_avg_and_min(self.tmp_solution)
    AGAMode = False
    if aga != EnumAGA.GA:
        if f_max != 0:
            AGAMode = True

        aga = self.popConfig.getExtraConfig().getAGA()
        AGAMethod = getattr(AGA, str(aga.name))

    dims = self.tunerConfig.dims
    size = self.popConfig.getSize()

    mutationOP = self.popConfig.getMutationConfig().getOP()

    # begin to matute
    for solution in self.tmp_solution:
        if AGAMode:
            rate = AGAMethod(self.popConfig.getMutationConfig().min_rate,
                             self.popConfig.getMutationConfig().max_rate,
                             f_max, f_avg, solution.score, target = 1)
        else:
            rate = self.popConfig.getMutationConfig().getRate()
        if np.random.random() < rate:
            if hasattr(GAMutation, str(mutationOP.name)):
                mutationMethod = getattr(GAMutation, str(mutationOP.name))
                gene = mutationMethod(solution.gene, self.tunerConfig.dims,
rate)
            else:
                gene = GAMutation.rm(solution.gene, self.tunerConfig.dims, rate)
            if self.popConfig.getExtraConfig().getSA() == EnumSA.GA:
                solution.gene = gene
                solution.score = 0.0
            elif self.popConfig.getExtraConfig().getSA() == EnumSA.GA_SA1:
                value = solution.score
                p = self.checkGene(self.T, value, solution.score)
                if np.random.random() < p :
                    solution.gene = gene
                    solution.score = 0.0
            elif self.popConfig.getExtraConfig().getSA() == EnumSA.GA_SA2:
                value = solution.score
                p = self.checkGene(self.T, value, (f_avg - f_min)/2)
                if np.random.random() < p :
                    solution.gene = gene
                    solution.score = 0.0
        next_solution.append(solution)

    self.solutions = next_solution
```

Population.checkGene

根据Metropolis准则返回接受新个体概率

```
def checkGene(self, T, f1, f2):
    f = (f1 - f2) / f2
    if f > 0:
        return math.exp(- f / T)
    else:
        return 1
```

Population.has_next

判断种群是否遍历完全

```
def has_next(self):
    return self.next_id < (self.popConfig.getSize() - 1)
```

Population.get_next

获取种群下一个体

```
def get_next(self):
    self.next_id += 1
    return self.solutions[self.next_id % len(self.solutions)]
```

Multipopulation

多种群GA，包含多个Population

下面说明其人工选择算子及移民算子

Multipopulation.eliteInduvidual

选取各种群中最优个体并保存

Multipopulation.immigrant

遍历各种群,其中最差个体被前一个种群的最优个体替代

tvm.autotvm.tuner

在tvm.autotvm.tuner.ga_tuner基础上改进的my_ga_tuner

tvm.autotvm.tuner.my_ga_tuner

包括类MyGATuner,下面说明其主要函数

MyGATuner.__init__

```
def __init__(self, task, pop_size=100, pop_num = 1, elite_num=3, iterate_num =
3,
             selection = EnumSelection.rws,
             crossover = EnumCrossover.spc,
             mutation = EnumMutation.rm,
             crossover_prob = 0.7,
             mutation_prob=0.1,
             update_rate = 0.3,
             aga = EnumAGA.GA,
             popsConfig = None)
```

当popsConfig为None时,则依据前置参数设置GA;

当popsConfig不为None时,则依据popsConfig的参数设置GA,此使前置参数无效,popsConfig可为dict或PopulationConfig或MultipopulationConfig类型.

当使用单种群GA时,可直接设置其各类参数,或使用popsConfig参数进行设置.

当使用MPGA时,则仅能使用popsConfig参数进行设置.

MyGATuner.next_batch

```
def next_batch(batch_size)
```

返回batch_size个个体

MyGATuner.update

```
def update(self, inputs, results):
    # 多种群移民
    self.multipop.update(inputs, results)
    self.multipop.eliteInduvidual()
    self.multipop.immigrant()

    judge = False
    # 根据迭代概率及所有种群是否已遍历完毕判断是否进行种群迭代
    for pop in self.multipop.pops:
        if np.random.random() < self.update_rate or not pop.has_next():
        # if not pop.has_next():
            judge = True
            break

    # 进行种群迭代
    if judge:
        for pop in self.multipop.pops:
            pop.update()

        self.trial_pt = 0
```

进行tuner种群迭代

MyGATuner.has_next

```python
def has_next(self):
    for pop in self.multipop.pops:
        if not pop.has_next():
            return False
    return True
```

判断tuner种群是否遍历完全