Lab 3 Local Search Algorithms

SUSTC

Problem Formulation for Specific Algorithms

- NQueensProblem is reformulated to fit the simulated annealing algorithm and the genetic algorithm. The code is given in “qnproblem.py”. And the problem class can be used by a line of code “from qnproblem import *”.
- In the reformulated NQueensProblem, a function sa_value( ) which calculates the number of pairs of queens attacking each other is used. This function is written in sa_function.py which should be put together with other codes in the same folder.

Problem Formulation for Specific Algorithms

```
def __init__(self, N):
    self.N = N
    "initial state is randomly generated"
    self.initial = random.sample(range(N), N)

def actions(self, state):
    "gengerate actions actd which contains all possible moves of each queen"
    actm=[[0] * (self.N-1)] * (self.N)
    for i in range(self.N):
        seq8=list(range(self.N))
        seq8.remove(state[i])
        actm[i]=seq8

    actd=[[0] * 2] * (self.N*(self.N-1))
    k=0
    for i in range(self.N):
        for j in range(self.N-1):
            actd[k]=[i,actm[i][j]]
            k+=1

    return actd

def result(self, state, actd):
    "move the queen in i_th col"
    new = state[:]
    for i in range(self.N):
        if actd[0]==i:
            new[i]=actd[1]

    return new

def value(self, state):
    "return 20 minus the number of pairs of attacking queen"
    return -sa_value(state)+20
```

Simulated Annealing Algorithm

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  for t = 1 **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if T = 0 **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next$.VALUE – *current*.VALUE
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Simulated Annealing Algorithm

- No revision, can be directly used from search.py.

```
def exp_schedule(k=20, lam=0.005, limit=100):
    "One possible schedule function for simulated annealing"
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)


def simulated_annealing(problem, schedule=exp_schedule()):
    "[Figure 4.5]"
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current
        neighbors = current.expand(problem)
        if not neighbors:
            return current
        next = random.choice(neighbors)
        delta_e = problem.value(next.state) - problem.value(current.state)
        if delta_e > 0 or probability(math.exp(delta_e / T)):
            current = next
```

```
q8problem = NQueensProblem(8)
myag=simulated_annealing(q8problem)
print(myag.state)
print(q8problem.value(myag.state))
```

Usage

Genetic Algorithms

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
   inputs: population, a set of individuals
           FITNESS-FN, a function that measures the fitness of an individual

   repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
         x ← RANDOM-SELECTION(population, FITNESS-FN)
         y ← RANDOM-SELECTION(population, FITNESS-FN)
         child ← REPRODUCE(x, y)
         if (small random probability) then child ← MUTATE(child)
         add child to new_population
      population ← new_population
   until some individual is fit enough, or enough time has elapsed
   return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
   inputs: x, y, parent individuals

   n ← LENGTH(x); c ← random number from 1 to n
   return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Original Genetic Algorithms

```
def genetic_search(problem, fitness_fn, ngen=1000, pmut=0.1, n=20):
    """
    Call genetic_algorithm on the appropriate parts of a problem.
    This requires the problem to have states that can mate and mutate,
    plus a value method that scores states."""
    s = problem.initial_state
    states = [problem.result(s, a) for a in problem.actions(s)]
    random.shuffle(states)
    return genetic_algorithm(states[:n], problem.value, ngen, pmut)


def genetic_algorithm(population, fitness_fn, ngen=1000, pmut=0.1):
    "[Figure 4.8]"
    for i in range(ngen):
        new_population = []
        for i in len(population):
            fitnesses = map(fitness_fn, population)
            p1, p2 = weighted_sample_with_replacement(population, fitnesses, 2)
            child = p1.mate(p2)
            if random.uniform(0, 1) < pmut:
                child.mutate()
            new_population.append(child)
        population = new_population
    return argmax(population, key=fitness_fn)
```

Revised Genetic Algorithms

```
def genetic_search(problem, ngen=100, pmut=0.5, n=4):
    # ______________________
    """
    Call genetic_algorithm on the appropriate parts of a problem.
    This requires the problem to have states that can mate and mutate,
    plus a value method that scores states."""
    # ______________________
    s = problem.initial
    # ______________________
    states = [problem.result(s, a) for a in problem.actions(s)]
    random.shuffle(states)
    return genetic_algorithm(states[:n], problem.value, ngen, pmut)

def genetic_algorithm(population, fitness_fn, ngen=100, pmut=0.1):
    "[Figure 4.8]"
    for i in range(ngen):
        new_population = []
        for i in range(len(population)):
            fitnesses = map(fitness_fn, population)
            p1, p2 = weighted_sample_with_replacement(population, fitnesses, 2)
#________________________________________________________________________________
            p1=GAState(p1)
            p2=GAState(p2)
#________________________________________________________________________________
            child = p1.mate(p2)

            if random.uniform(0, 1) < pmut:
                child.mutate()
            new_population.append(child.genes)
        population = new_population
    return argmax(population, key=fitness_fn)
```

Input para changed

Var name changed

p1 and p2 are transformed to the genetic state

Revised Genetic Algorithms

```
class GAState:
    "Abstract class for individuals in a genetic search."
    def __init__(self, genes):
        self.genes = genes

    def mate(self, other):
        "Return a new individual crossing self and other."
        c = random.randrange(len(self.genes))
        return self.__class__(self.genes[:c] + other.genes[c:])

    def mutate(self):
        "Change a few of my genes."
        # _________________________________
        ind_mute=random.randrange(len(self.genes))
        self.genes[ind_mute]=random.randrange(len(self.genes))
        #raise NotImplementedError
        # _________________________________
```

Mutate function added

Revised Genetic Algorithms

```
q8problem = NQueensProblem(8)
myag=genetic_search(q8problem)
print(myag)
print(q8problem.value(myag))
```

Usage

Conclusion

- “qnproblem.py” and “sa_function.py” are to be downloaded and put together with your other codes in the same folder. simulated_annealing( ) can be directly used by importing from “search.py”, and genetic_search( ) should be revised according to the above text before being used.