## Genetic algorithms in simpleai

• local.py contains the genetic algorithm in simpleai

you need to look at the functions genetic(), \_local\_search()
 and \_create\_genetic\_expander()

they very much follow the theory in chapter 4 of the textbook,
 and in my other write-up in this module 11

• so, we will just apply the theory to assignment 5

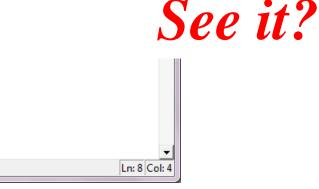
## **Assignment 5**

- the assignment is solving a KP (Knapsack Problem) using a genetic algorithm
- you must use simpleai, of course, and exactly the following data:
  - ♦ 20 objects
  - $\bullet$  weights of objects w = [4, 6, 5, 5, 3, 2, 4, 8, 1, 5, 3, 7, 2, 5, 6, 3, 8, 4, 7, 2]
  - $\bullet$  values of objects v = [5, 6, 2, 8, 6, 5, 8, 2, 7, 6, 1, 3, 4, 4, 1, 5, 6, 2, 5, 3]
  - lacktriangle maximum weight the knapsack can carry = 35
- before I say something about the functions involved, I show what the program outputs on several runs (in Canopy or otherwise)

A

This run has found the solution consisting of the 1<sup>st</sup>, 2<sup>nd</sup>, 4<sup>th</sup>, 5<sup>th</sup>, 7<sup>th</sup>, 9<sup>th</sup>, 10<sup>th</sup>, 13<sup>th</sup>, 16<sup>th</sup>, and 20<sup>th</sup> objects, whose total weight is 35, and total value is 58

Remember that a *local search*, like the *genetic algorithm*, is <u>not</u> guaranteed to find an *optimal* solution



```
76 Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win
Type "copyright", "credits" or "license()" for more information.
[('crossover', [1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1])]
Value = 58
[('crossover', [1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1])]
Weight = 35
Value = 56
>>>
                              A next run may even produce a
                              solution worse than before, as
                              shown by my second run
```

Ln: 13 Col: 4



other

• now, review module 10, if you have forgotten what KP

(Knapsack Problem) is, and how a solution (a state for us)
is represented by a list of 0s and 1s, where 1 signifies an
object is selected, and 0 signifies an object is not selected
to be carried in the knapsack [the DNA is that list of 0s and 1s]

although there are many ways of handling these bit patterns
(lists of 0s and 1s) in Python, for now we will just apply
what we know from previous modules

• review previous modules as necessary

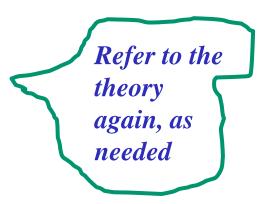
• just as with previous assignments using *simpleai*, make a **KnapsackProblem** class, that sub-classes *SearchProblem* 

 the \_\_init\_\_() constructor is passed values for the instance variables numObjects, maxWeight, weights, values

• numObjects is the number of objects, to be 20 then when the class is instantiated

• maxWeight is the maximum weight the knapsack can carry

- weights and values are lists [the actual lists to be eventually used are shown on a previous slide] that provide the respective weights and values of the objects
- now, by the requirements of the *simpleai* API, your class must provide the following 'public' methods, to use the genetic algorithm:
  - **♦ generate\_random\_state()** [random DNA]
  - ♦ crossover()
  - mutate()
  - ♦ value() [that is, fitness]



• but first, I want you to provide 2 'private' methods in the class, useful in the above methods as well

one is

```
def _weight(self, s)
''' returns the total weight of the objects in the selected state s '''
```

this should be trivial, looping through range(0, len(s))
 and using self.weights

 remember that s is a list of 0s and 1s telling you which objects are in s the other 'private' method is

```
def __valid(self, s)
    ''' returns True if choice s is valid; that is,
        total weight of s <= self.maxWeight
'''</pre>
```

easy, using the previous 'private' method \_weight()

the 'public' method

```
def value(self, s)
    ''' returns the total value of the objects in the selected state s '''
```

is straightforward too now, using the list self.values

now, for the public 'method'

```
def generate_random_state(self)
    ''' returns a valid random bit pattern (list of 0s and 1s)
        corresponding to a subset of the objects to be carried
    '''
```

an idea is to start off with all objects chosen, which will
probably be not a valid choice [a valid choice is where the
total weight is <= self.maxWeight; see above 'private' method]</li>

• in Python, note that you may make a list like

```
choice = [1] * self.numObjects
```

ah, you know that, as concatenation,

[1, 1] is the same as [1] + [1] in Python which should be the same as [1] \* 2 by algebra then

• hence, [1] \* 5 is [1, 1, 1, 1, 1]

• likewise, [0] \* 6 is [0, 0, 0, 0, 0, 0]

- likewise, [5, 3] \* 4 is [5, 3, 5, 3, 5, 3, 5, 3]
- get it?

• also, recall that range(0,N) is the list [0, 1, 2, ..., N-1]

- so, for generate\_random\_state() for KP, you may loop as long as you get an invalid choice of objects, doing the following:
  - 1. choose a random subset size k (between 1 and self.numObjects)
  - 2. choose a random subset x of size k in range(0, numObjects)
  - 3. deselect the objects whose index is not in x

• return the valid choice of subset you get

in other words, you may structure generate\_random\_state()
 as follows

```
def
     generate_random_state(self)
      returns a valid random bit pattern (list) for a subset of the objects to be carried
   r = range(0, self.numObjects)
   choice = [1] * numObjects
   while not self._valid(choice):
       # select random size k, from 1 to self.numObjects, using random.randint()
       # select random subset x of length k from r, using random.sample()
       # change choice[i] to 0 for those not in x
                                                            These are standard
  # after loop
```

return choice

now, for the next 'public' method

```
def crossover(self, s, t)
''' returns a valid crossover of state s and state t'''
```

- this should be straightforward
- you select a random index k (between 1 and self.numObjects -1;
   indexing on a list is 0-based) using random.randint()
- using slicing, you may certainly concatenate the part s[:k] in s, with the part t[k:] in t, to get some crossover y, right?
- except that the above may not be a valid crossover

so, you need to loop

```
while not self._valid(y):
    ''' repeat the making of k and y '''
```

• your method returns the valid y you get

• very simple, but all to mediate about

• now, there might be a problem, that might get your program into an infinite loop

 you may keep repeating doing the same k and same y, and it will never end if you never get a valid choice

• but, there is no reason to choose a k more than self.numObjects times, right?

so, keep a count, and if you do the loop more than
 self.numObjects times, then break the loop and simply return s,
 not a crossover of s and t, because you found none that is valid

something to think about

now, for the last 'public' method

```
def mutate(self, s)
''' returns a valid mutation (another state) of state s ''
```

 you would choose a random index n, between 0 and self.numObjects - 1 where you will mutate (switch)

• at position n, switch s[n] to 0 if it is 1, to 1 if it is 0 [that sounds like a mutation; but, certainly not the only choice]

• however, as with crossover() there are problems to care about

• changing 1 to 0 alone is bad, since you just *lower* the weight (s is already valid to begin; removing an object makes it worse)

• you <u>must</u> follow this change 1 -> 0 with another random change 0 -> 1 somewhere else, to improve it, right?

• changing 0 to 1 might produce an *invalid* combination of objects, exceeding self.maxWeight

 so, you need to check if the choice you get is valid (produce only a valid mutation of s)

## here is how you might structure your method then

```
def
     mutate(self, s)
      returns a valid mutation (another state) of state s ""
                                                                Do not forget to
   valid = False
                                                                reset that valid
   n = -1
                                                                flag, as needed
   while not valid:
      # choose your random n between 0 and self.numObjects - 1
      if not s[n]:
                             \# s[n] is 0
           # do what?
                                                  As in the case of crossover(), you
                             \# s[n] is 1
     else:
                                                  may get into an infinite loop,
          # do what?
                                                  keeping doing the same n
  # after all that
                                                  So, keep a count, and do not do
  return s
                                                  the loop more than
                                                  self.numObjects times anyway
```

• the executable (script) part of your program must of course instantiate your **Knapsack** class, call the **genetic()** algorithm, and print relevant outputs

something like

```
Do read your simpleai
\mathbf{0} = \dots
maxw = ...
\mathbf{w} = \dots
\mathbf{v} = \dots
problem = Knapsack(o, maxw, w, v)
result = genetic(problem, iterations_limit=100, population_size=16, mutation_chance=0.10)
print result.path()
print 'Weight = ' + str(problem._weight(result.path()[0][1]))
print 'Value = ' + str(problem.value(result.path()[0][1]))
```