6G6Z1109: Software Agents and Optimisation

Term 2, Lecture 8: Assignment, and comparative analysis

Assignment help

- Your solution will require classes to represent a GA individual, the population, and the ordered crossover operator (as with the TSP solution)
- It also needs the classes for the circle placement algorithm (`Bunch`, `Circle`, and `Point`). You should not need to make many changes to this code…
- The main application should create a `Bunch` object for *each* of the three algorithms – this allows you to easily display all three

Assignment help

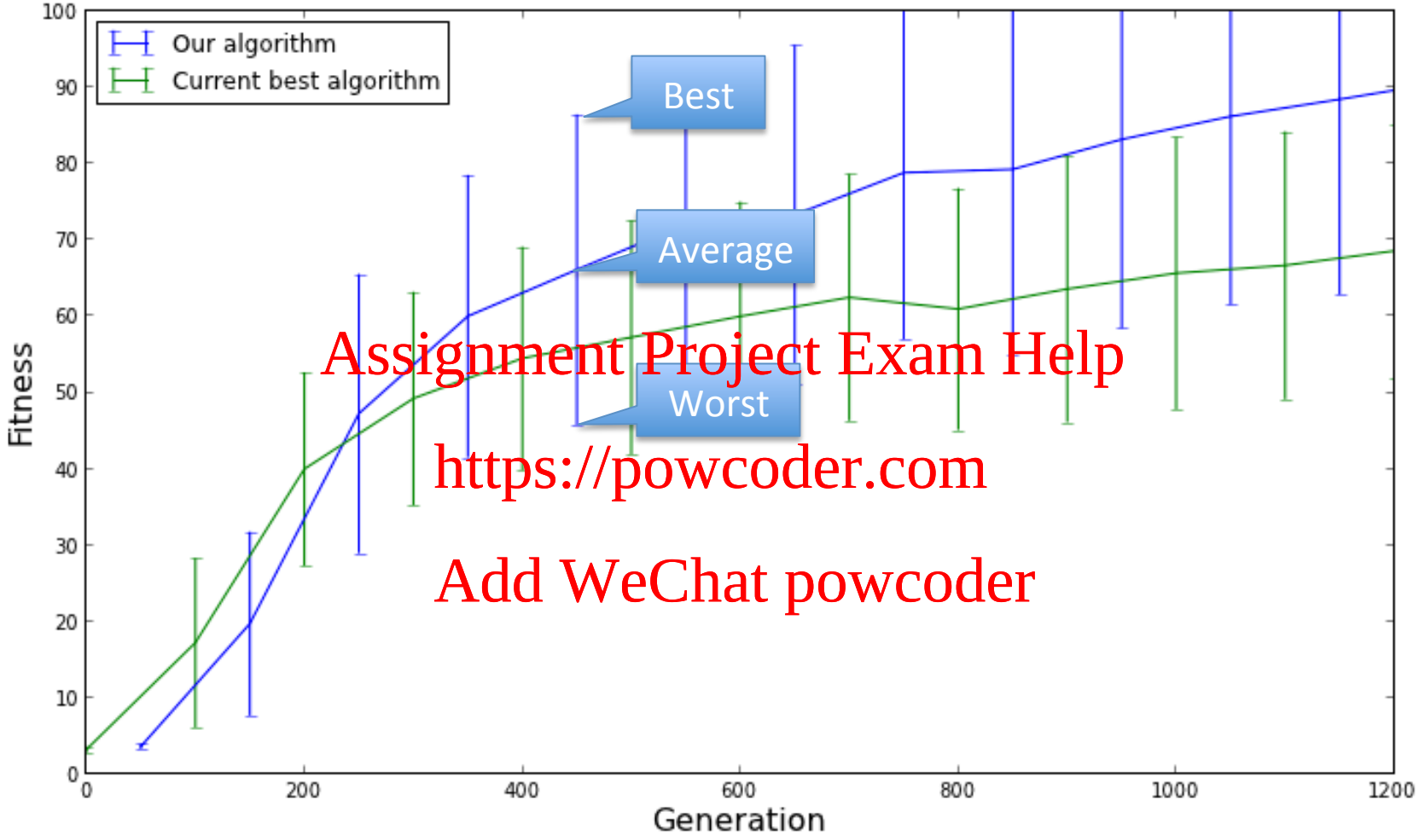
- Your `Bunch` class should contain a `geneticPlace()` method, which takes in all of the parameters of the GA (that is, it should be entirely self-contained)
- It will be *much* easier to implement if you have already successfully implemented the `Population` class (as in a previous lab)
- Given that we are using an order-based encoding, it's no surprise than an individual's genome should be a list of integers – the ordering of the set of circles
- That is, a genome of "21345", specifies circle 2 to be placed first, followed by 1, 3, 4, then 5

Assignment help

- When evaluating an individual's genome in the fitness function, you should realise that you can actually reuse the `orderedPlace()` method that we have already encountered
- You can *overload* this method, by providing a version that *takes in* an ordering (encoded as an array of integers…) and places the circles in that order
- The fitness function in my model answer (located in the `Individual` class) is *extremely short*; it simply creates a temporary `Bunch` of circles, orders them according to the genome, calls the method to compute the boundary, and then returns this as the fitness…

Assignment help

- When doing a *comparative analysis* of your genetic algorithm (i.e., comparing with greedy search), it is important to specify the parameters used, and investigate the effect of varying these (see previous lecture)
- Also, it is important to perform *several* runs of your GA, and then take the *average* best fitness found, in order to ensure that you get a representative sample
- How many runs is "enough"? – probably around 30-50
- It is also interesting to plot the *convergence* of your algorithm over time



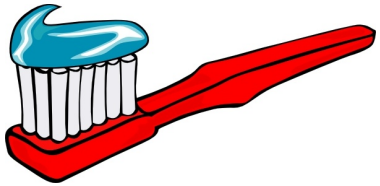
R. Olson

Encodings

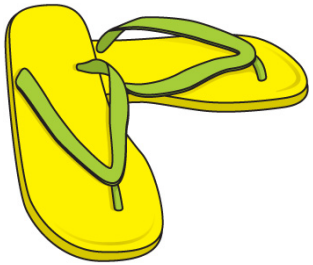
- How we *encode* a solution to our problem is also vital to the success of the GA
- Ideally, our encoding will be expressed as a linear *string* or sequence of items, such as characters, integers, reals, etc.
- An encoding might *directly* represent the objects that are evolving (e.g. strings of text), or *indirectly* represent the presence or absence of objects with certain attributes

Indirect encoding example

- Consider the following problem: we wish to pack for our holiday, and we have a suitcase with a specific capacity (luggage weight limit)
- We have a set of *items*, each of which has a *weight* and a *utility* value (that is, how *useful* we find it)
- We wish to find the *set of items* that maximises our utility, whilst staying within the weight limit



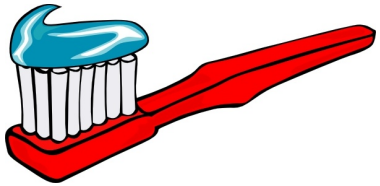
Utility: 8
Weight: 4



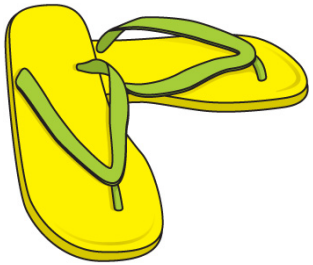
Utility: 5
Weight: 5

Utility: 9
Weight: 7

Utility: 10
Weight: 5



Utility: 8
Weight: 4



Utility: 5
Weight: 5

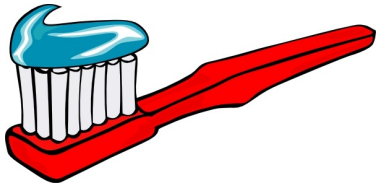
Utility: 9
Weight: 7

Utility: 10
Weight: 5

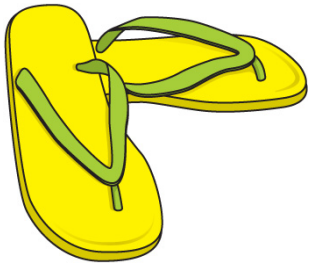
How do we encode a solution to this problem?

Choosing an encoding

- We need to work out what the problem is asking of us…
- It needs us to find a *subset of items*, subject to certain constraints
- That is, each item is either "in" or "out" of the suitcase
- A solution is a *candidate list of items*, which is then assessed according to the criteria
- In this case, if we *order* the items, a simple *binary encoding* will suffice



Utility: 8
Weight: 4

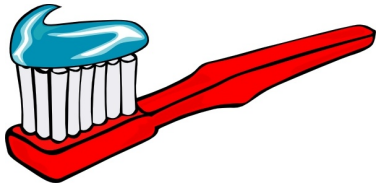


Utility: 5
Weight: 5

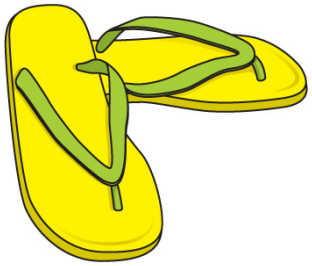
Utility: 9
Weight: 7

Utility: 10
Weight: 5

We can now use a simple binary encoding on the ordered list of items, so 1001 represents “include toothbrush and suncream”, whilst 0111 represents “flip-flops, bikini and suncream”



Utility: 8
Weight: 4



Utility: 5
Weight: 4

Utility: 9
Weight: 7

Utility: 10
Weight: 5

We call a *sequence* the *genotype*, as it represents an individuals "genetic code". What a genotype represents or builds is called the *phenotype,* and it is this "body" that we assess or assign a fitness value to.

A *greedy* algorithm

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

A greedy algorithm

Profit/weight

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

A greedy algorithm

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

Item	Profit	Weight	PTWR
1	6	2	3
2	5	3	1.67
3	8	6	1.33
4	9	7	1.29
5	6	5	1.2
6	7	9	0.78
7	3	4	0.75

Capacity=9

Space=9

A greedy algorithm

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

Item	Profit	Weight	PTWR
1	6	2	3
2	5	3	1.67
3	8	6	1.33
4	9	7	1.29
5	6	5	1.2
6	7	9	0.78
7	3	4	0.75

Capacity=9

Space=7

A greedy algorithm

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

Item	Profit	Weight	PTWR
1	6	2	3
2	5	3	1.67
3	8	6	1.33
4	9	7	1.29
5	6	5	1.2
6	7	9	0.78
7	3	4	0.75

Capacity=9

Space=4

A greedy algorithm

Calculate the *profit-to-weight ratio* for each item
While (we can select an item)
    Select item with highest PTWR that keeps us below capacity

Item	Profit	Weight	PTWR
1	6	2	3
2	5	3	1.67
3	8	6	1.33
4	9	7	1.29
5	6	5	1.2
6	7	9	0.78
7	3	4	0.75

Capacity=9

Space=0

Total profit: 14

A genetic algorithm

We use a simple binary encoding scheme, with a binary vector x=[x_1, ..., x_n] to represent n items, where x[i]=1 if the item is *included* in the suitcase, and x[i]=0 if it is not

The fitness function for a given solution, x, is as follows:

We denote the profit of items by p[i], and the weights by w[i]


```
total weight = 0
for each item, x[i]                          // weigh items in suitcase
    if x[i] = 1 then add w[i] to total weight

if total weight <= total capacity
    fitness = 0
    for each item, x[i]                      // add profits together
        if x[i] = 1 then add p[i] to fitness
    return fitness
else                                         // over capacity
    return capacity-total weight
```

A genetic algorithm

We use a simple binary encoding scheme, with a binary vector x=[x_1, ..., x_n] to represent n items, where x[i]=1 if the item is *included* in the suitcase, and x[i]=0 if it is not

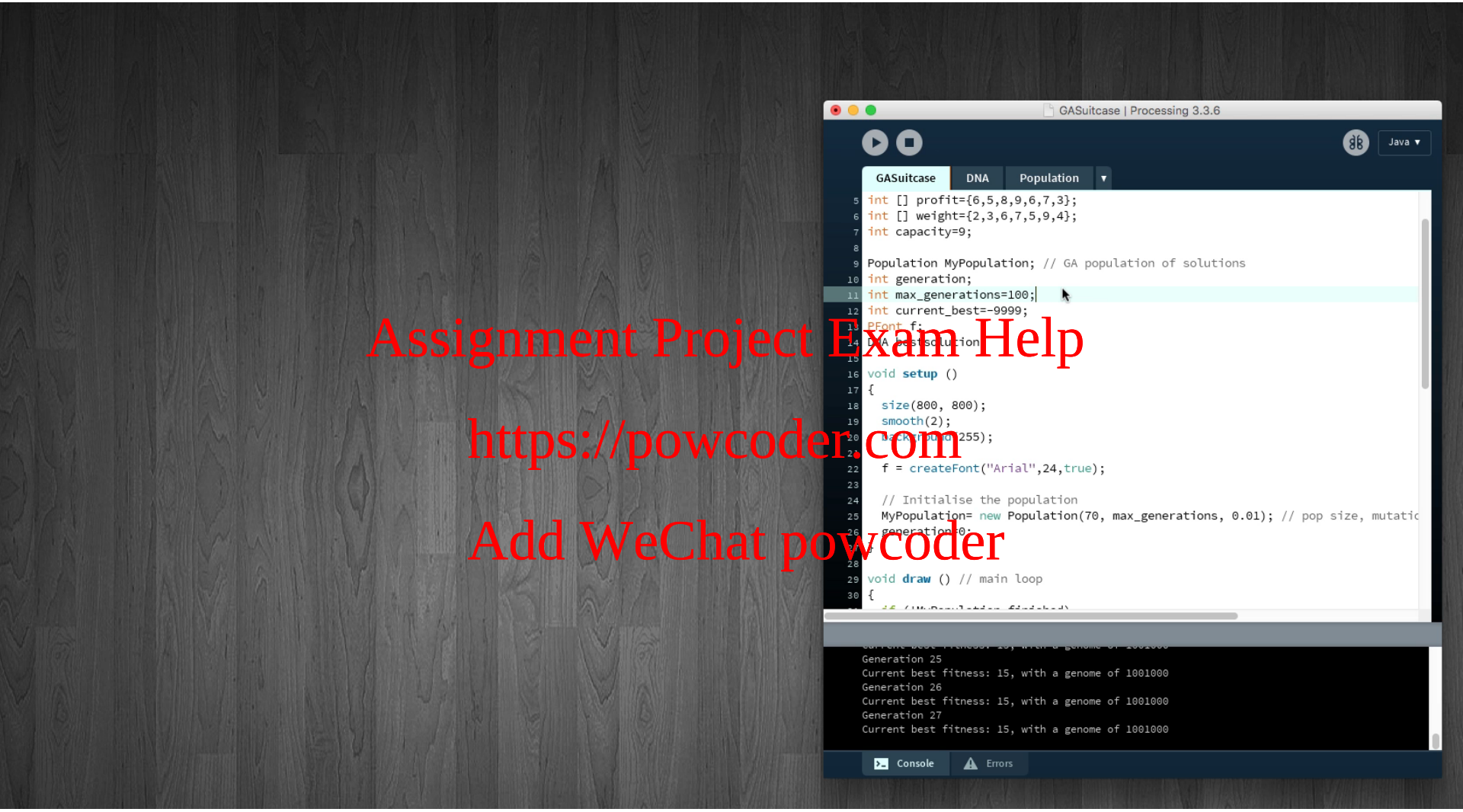
The fitness function for a given solution, x, is as follows:

We denote the profit of items by p[i], and the weights by w[i]


```
total weight = 0
for each item, x[i]                          // weig
    if x[i] = 1 then add w[i] to total weight

if total weight <= total capacity
    fitness = 0
    for each item, x[i]                      //
        if x[i] = 1 then add p[i] to f      ness
    return fitness
else                                         // over capacity
    return capacity-total weight
```

Penalise “illegal” solution, by essentially giving it a negative fitness


Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder
GASuitcase | Processing 3.3.6
Java
GASuitcase
DNA
Population
int [] profit={6,5,8,9,6,7,3};
int [] weight={2,3,6,7,5,9,4};
int capacity=9;
Population MyPopulation; // GA population of solutions
int generation;
int max_generations=100;
int current_best=-9999;
PFont f;
DNA bestSolution;
void setup ()
{
size(800, 800);
smooth(2);
f = createFont("Arial",24,true);
// Initialise the population
MyPopulation= new Population(70, max_generations, 0.01); // pop size, mutatic
generation=0;
void draw () // main loop
{
Generation 25
Current best fitness: 15, with a genome of 1001000
Generation 26
Current best fitness: 15, with a genome of 1001000
Generation 27
Current best fitness: 15, with a genome of 1001000
Console
Errors

Next lecture

- Next week: (Remember, *this lecture is next Monday*) - advanced topics in GAs
- This week's lab: Knapsack GA (but only move onto this once you have finished your assignment!) This lab is still important, though…