

Assignment #4 – Parallel Shakespearean Monkeys

Part II – version 2

Specs

You are required to build a Java Swing application that illustrates the Shakespearean Monkeys genetic algorithm, taking the given demo as a model (which is adapted from an MSDN parallel extensions demo).

This application functions in two distinct modes:

1. **Sequential** mode.

You can fully design and implement this using

- the topics learned in parts 1 (**OO**) and 2 (**Swing**) of this course and
- the 1st handout of part 3 (**Threading**), which is *only* required to keep your application **responsive**

2. **Parallel** mode.

This extends your design with a **parallel** implementation, using

- topics learned in the 3rd handout of part 3 (**Threading**) of this course

If properly designed, the two execution modes **share most of the code** and only differ in their implementation of a critical loop.

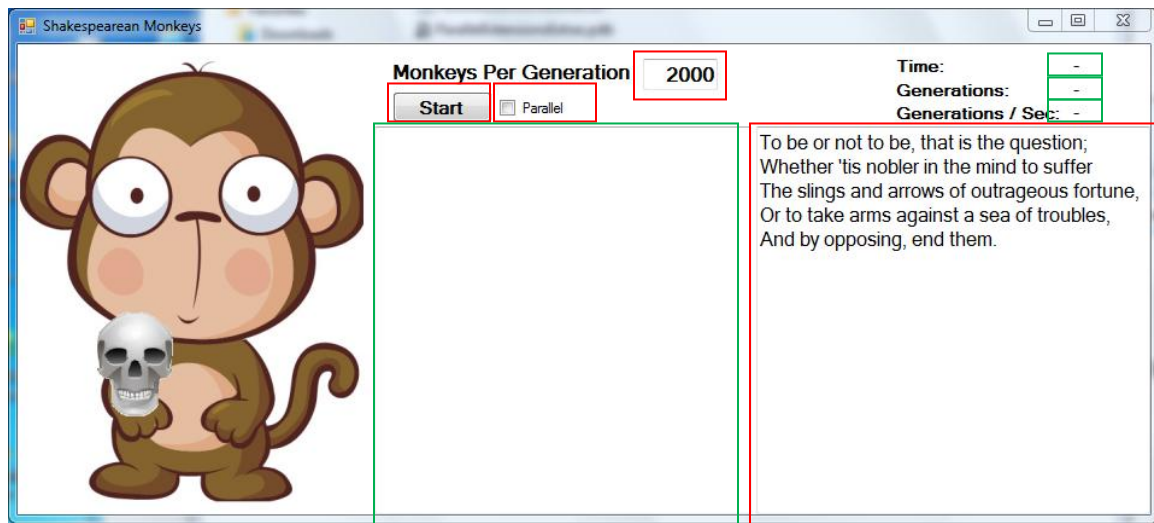
Bird's eye view

This genetic algorithm **simulates** the evolution of a population of “monkeys”. To properly model a real life evolution, the simulated model must follow several rules. Here we give a bird's eye view; further details are described in the following pages.

- The model evolves via successive **generations** (this is the main loop)
- At each evolution step, the current generation (“parents”) creates an **equally sized** new generation (“children”)
- At the end of the evolution step, the new generation **becomes** current and the old generation is discarded
- Each monkey receives a **fitness score**, indicating how close its genome is from the target. The fitness score is a single number, indicating the total number of differing positions. It is **not** allowed to consider the actual positions where the genomes differ!
- The **breeding** process which builds the new generation (this is the most critical inner loop) gives higher priority to monkeys with higher fitness scores; i.e. these are more likely to be selected as parents
- The parent selection is with **repetition**. A high scoring monkey can be selected parent in a large number of parent pairs
- Although this will very rarely happen in a large population, the same monkey can be accidentally selected for **both** parents. In this case, the two children are identical to this parent, regardless if a crossover occurs or not. This does not statistically affect the final outcome and the simulation code is simpler and faster
- However, even the lowest scoring monkey still has a small but not null **chance** to contribute to the next generation (further details later)
- The evolution stops when one of the new monkeys has **exactly** the target genome (i.e. having the maximum fitness score).

GUI

The GUI must be a reasonably close match to the demo GUI (except the picture which could be replaced by your favourite pet):



The following controls, highlighted by red contours, define algorithm *inputs* and *control* its behaviour:

- **PopulationSize** (initially 2000): the number of monkeys in a generation (editable)
- **TargetText** (initially Hamlet's stance): the target text (editable)
- **Start** button (initially caption **Start**): the first click changes its caption to **Cancel** and starts the evolution; the second click (now using caption **Cancel**) stops the evolution (regardless if the evolution runs sequentially or in parallel)
- **Parallel** checkbox (initially unchecked): if checked, requests a parallel evolution; otherwise, requests a sequential evolution

The following controls, highlighted by green contours, define algorithm *traces* (including its final *output*):

- **Time** (initially -): total time since evolution start (secs)
- **Generations** (initially -): number of generations
- **Generations / Sec** (initially -): average number of generations per second (i.e., evolution speed)
- **Best current text** (initially blank): a sequence of best generated texts; the last one, which matches the target text, should be highlighted by a greenish background colour

Evolution rules

Clicking on the **Start** button must also *validate* the user inputs:

- The **PopulationSize** box must contain a number in the range 1 to 10,000
- The **TargetText** must contain a text of length in the range 1 to 300 and all characters must be in the set {**10, 13, 32-126**}, i.e. NL='\n' (new line), CR='\r' (carriage return) plus all printable 7 bit ASCII characters (including space)
- Errors panels must be raised if these conditions are not met

Rules for *updating* the trace controls:

- The trace controls are reset to their initial values each time a new evolution starts (i.e., after a click on the **Start** button).
- Then, as generations evolve very fast, the trace controls should *only* be updated when the current best text is **strictly better** than the previous (it is possible that, at times, the new generation performs worse or no better than its parent, in which case *no* update should occur).

Responsiveness

- The application must remain **responsive** at all times (thus the Start event handler must start at least one background thread even for sequential evolutions, and, of course, more for the critical parts of the parallel evolution)
- In particular, a click on the **Cancel** button (button **Start**'s other face) must completely cancel the evolution as soon as possible and revert to its initial caption, Start

Major assessment criteria

Programs which compile and run as expected will be further differentiated according to the following criteria:

- On typical lab machines (4-8 logical cores), the parallel version should run faster than the sequential version
- The parallel version should run faster than **demo v1** (because this **demo v1** is not optimised) and the task manager should show **high CPU utilisation** (all logical cores are busy)
- The source code is **readable** and **well structured**
- The coding **differences** between the sequential and the parallel modes are well **localised**; most of the lines should be **common** between the two modes
- The GUI updates are performed in a **thread-safe** manner (i.e. a background thread can legally update only (1) using **SwingUtilities**, or (2) using explicitly declared (see API) **thread-safe** controls and methods (such **JTextComponent**'s **setText**)
- The background threads are properly "cancelled" (some form of cooperation seems needed)
- There are **no race** conditions between the background thread (you can **avoid** these, but, if they appear, use proper locks or concurrent data structures)
- Of course, your code must **not** "cheat" by identifying matching positions: you can only base your evolution on a single fitness score

Last but not least

- Please read the note on **Plagiarism and Cheating** in our Assignments page.

Pseudocode

Internally, the code uses the following additional input parameters:

- **CrossoverProbability** (default 0.87): the probability that a pair of parents will cross over their genomes while generating a pair of children; otherwise, the children are identical to their parents
- **MutationProbability** (default 0.02): the probability that a newly generated child will suffer one random mutation

We recommend that the expected algorithm should follow the following high level pseudocode (this is just a hint, you can do better, if you want):

```
// to keep the app responsive, run all this in a background thread (not in GUI/EDT)

create the initial random population (strings or arrays of valid characters, each one
same size as the target text size, i.e. TargetText.Length)

evolution loop
  cancel if cancellation was required
  compute fitness scores and find best match of the current generation
  update the GUI if the new best is strictly better than the previously displayed best
  break if the best text matches the target text

// create new generation
repeat PopulationSize/2 times, sequentially or in parallel
  // select two parents from current generation and create two children
  p1 = random parent, giving more chances to high quality monkeys
  p2 = random parent, giving more chances to high quality monkeys
  if random < CrossoverProbability then
    CrossoverIndex = random between 1 and p1.Length
    c1 = 1st part of p1 + 2nd part of p2
    c2 = 1st part of p2 + 2nd part of p1
  else
    c1 = p1
    c2 = p2
  if random < MutationProbability then
    randomly change one single char in c1
  if random < MutationProbability then
    randomly change one single char in c2
  add c1 and c2 to the new generation
goto repeat

current generation = new generation
// you can recycle the storage of the previous current generation
goto evolution loop
```

The **red** highlighted words indicate critical parts.

- **update the GUI**: do this properly, because you are in a background thread
- **repeat**: the single most efficient point to swap between a **sequential** and a **parallel** execution mode; there are a few other places, but the cost/benefit will be less significant; the demo introduces parallelism only at this point
- **add** c1 and c2 to the new generation: in the parallel mode, you **could** have a **race** condition, as several threads could try to update the new generation at the same time; however, you can also design your tasks such that **no** race will ever occur here
- the next discussion suggest one good way to give **more chances to high quality** parents (it is a simple well-known statistical procedure)

How to select parents

This is best explained by an example. Consider that we have a target text of length 10 and four monkeys, m[0], m[1], m[2], m[3], with 3, 8, 3, 5 matching characters (respectively), as indicated in the following table:

Monkey	Text	Matching count	Difference count	Breeding weight
m[0]	...	3	7	1
m[1]	...	8	2	6
m[2]	...	3	7	1
m[3]	...	5	5	3

The highest matching score, 8, is achieved by m[1], which can also be viewed as the smallest difference to the target, $2 = 10 - 8$. The largest difference, 7, is achieved m[0] and m[2].

For an efficient breeding, monkeys should receive the breeding weights as indicated in the last column, i.e. computed by the following formula:

$$\text{current breeding weight} = (\text{largest difference} - \text{current difference} + 1).$$

With this formula, m[1] gets the highest breeding weight, while m[0] and m[3] the lowest (but, because of the “+1” term, they can still marginally contribute).

The following pseudo-code shows how to randomly select a high quality parent:

```
sum-of-weights = sum of breeding weights (11 in our case)
val = random number between 0 and sum-of-weights-1 (0, 1, 2, ..., 10)
for i = 0 to PopulationSize-1 do
  weight = breeding weight of m[i]
  if val < weight then
    select m[i] as parent
    break
  val = val - weight
end for
```

Demo

The **demo (v1)** is a .NET program, slightly adapted from a MSDN parallel extensions sample, which is compiled for the .NET 4.5 platform, as installed in the labs.

If you want to experiment it on other machines, you need .NET 4.5, which is best achieved by installing VS 2012 Professional (or higher).

As student in computer science, you can download and install at home a free non-expiring version, to use it for university related studies.