Assignment #2.3 – Shakespearean Monkeys

Specs

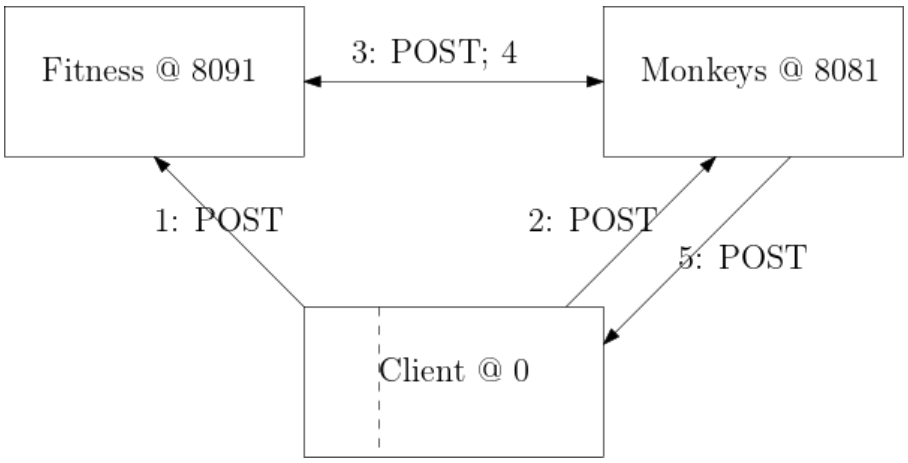
You are required to build a distributed application that illustrates the Shakespearean Monkeys genetic algorithm.

A#2.3 is based on A#2.2 and requires an extended version of server **Monkeys**. You are free to reuse snippets from your own A#2.2, from the model A#2.2, from the given skeletons, or any combination of these.

For the **required** part, the **Fitness** and the **Client** components will come for **us**, so you need not bother about these, except closely following the required wire protocols (REST, JSON). Also, there is a **bonus** for creating your own **Client** as a **Web application**.

In the following diagrams, a bidirectional arrow indicates POST requests (3) and responses (4). Unidirectional arrows indicate POST requests with empty OK responses (1, 2, 5).

Also, all *strings* (target, genomes) exclusively consist of characters in the range 32..126 (i.e. printable ASCII).



Marks:

- Basic part, sequential mode, given target length: 8 marks
- Parallel mode: 1 mark
- Dynamic length discovery: 1 mark
- Bonus: See last page

Submission: to **ADB**. Submit one single **jbon007-monekys.cs** file, containing everything needed to build and run **Monkeys**. The markers will use their own **.csproj** file, that only includes **Carter** and **Microsoft.AspNet.WebApi.Client** – do NOT use anything else.

There are a few significant differences in the wire protocols (JSON):

1. **POST /target** sends a JSON object, formatted as:

   { “id”: *int*, “parallel”: *Boolean*, “target”: *string* }

   Empty OK response.

2. **POST /try** sends a JSON object, formatted as:

   { “id”: *int*, “parallel”: *Boolean*, “monkeys”: *number*, “length”: *number*,
   “crossover”: *number*, “mutation”: *number* , “limit”: *number* }

   length > 0 indicates the actual target length

   length = 0 indicates that this length must be **dynamically discovered**

   crossover and **mutation** indicate probabilities as percents, e.g. 87 indicates 0.87.

   limit, with default 1000, is the maximum number of generations that will be created by the evolution loop (this stops even if the top doesn’t achieve the perfect score 0)

   Empty OK response.

3. **POST /assess** includes an array (cf. C# list) of JSON *strings* (genomes), one for each monkey, such as:

   { “id”: *int*, “genomes”: [“…”, “…”, … “…”] }

   This request must be followed by a matching non-empty response (4).

4. This POST response to (4) includes an array (cf. C# list) of numbers (fitness scores), one for each sent genome, in the same **order**:

   { “id”: *int*, “scores”: [*number*, *number*, … *number*] }

5. **POST /top** sends a JSON object, formatted as:

   { “id”: *int*, “score”: *number*, “genome”: *string* }

   Genome is one of the top-ranking genomes for the current generation, and score its Fitness score. Empty OK response.

Steps (3, 4, 5) are repeated until a genome reaches the ideal score = 0.

All POST bodies start now with an **id**, which is the port number of the client. With this convention, Fitness and Monkeys can support several concurrently running clients, each with its own port, targets, and requests (the default client port is 8101).

Each server runs, independently, in two distinct modes: (1) sequential mode, (2) parallel mode. If properly designed, the two execution modes **share most of the code** and only differ in their implementation of a critical loop. Attention at the parallel results (4), which must be returned in the same **order** as the received genomes.

Bird's eye view (Monkeys)

This genetic algorithm simulates the evolution of a population of "monkeys", until they learn to reproduce specific target text. 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 initial generation consist of randomly generated genomes, all of the given target length.
- If length = 0, Monkeys will have to discover the target lengths by trial and error.
- 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.
- All genomes of the current (or initial generation) are sent to **Fitness** for evaluations.
- Each genome receives a score, indicating how close it is from the target. The score is the Hamming distance between the target and that genome. Thus, **a lower score indicates a better fit**.
- The breeding process which builds the new generation (this is the most critical inner loop) gives higher priority to genomes with lower scores (higher fitness); i.e. these are more likely to be selected as parents.
- The parent selection is with repetition. A low scoring (high fit) genome can be selected parent in many parent pairs.
- Although this will very rarely happen in a large population, the same genome 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 outcome and the simulation code is simpler and faster

- However, even the high scoring (low fitting) genomes still have a small but not null chance to contribute to the next generation.
- The evolution stops when one of the new genomes fully matches the target (i.e. has score = 0).

We recommend that your algorithm follows the following high-level pseudocode (this is just a hint; you could perhaps do better):

```
current generation =initial random population: strings of the given (or inferred) length,
consisting of printable ASCII characters

evolution loop
   compute scores for the current generation - using Fitness, POST & response
   POST to Client, if the new best is strictly better than the previously displayed best
   break if the best genome fully matches the target text (score = 0)

   // create new generation
   repeat PopulationSize/2 times, sequentially or in parallel
      // select two parents from current generation and create two children
      p1 = random parent, chances proportional to genome's fitness
      p2 = random parent, chances proportional to genome's fitness
      if random < CrossoverProbability then
         CrossoverIndex = random index between 0 and length-1
         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
   end repeat

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

The above pseudo-code uses the following additional input parameters:

- CrossoverProbability (e.g. 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 (e.g. 0.02): the probability that a newly generated child will suffer one random mutation

How to select parents

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

Genome	Matching count	Hamming score (from **Fitness**)	Breeding weight
m[0]	3	**7**	1
m[1]	8	2	6
m[2]	3	**7**	1
m[3]	5	5	3

The highest fit, 8, is achieved by m[1], which has the smallest Hamming score to the target, 2. The **largest Hamming score**, **7**, is achieved by m[0] and m[2].

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

current breeding weight = (**largest Hamming score** – current Hamming score + 1).

With this formula, m[1] gets the highest breeding weight, while m[0] and m[2] the lowest (but, because of the "+1" term, they can still occasionally contribute, sometimes quite effectively).

The following pseudo-code shows how to randomly select a high fit parent, but still give chances to all others:

```
// weights-based selection

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
```

Suggestion

Start developing Monkeys' genetic algorithm as a **simple command-line app** (no client/server role). For this, you will need a local copy of the target, a copy of the Fitness evaluation function, then possibly get inputs from stdin, write outputs to stdout.

For a quick jump start, use the provided **Monkeys-standalone skeleton** project.

When this algorithm works perfectly, start to gradually add the client/server roles. Otherwise, errors that happen in a distributed app are often quite difficult to understand, to localise, and to fix.

Bonuses

- +1: for a browser app playing the client role
- +1: if this browser app uses SignalR
- +1: if this browser app uses C# Blazor, i.e. WASM, instead of client-side JS

The bonus must faithfully play the client role, exactly as our command-line client app. It is not allowed to become a monolithic app, including its own fitness and monkeys components.

There will just one single bonus submission. So, if you attempt all three bonuses, they must be implemented by the same bonus client app.

The bonus will only be considered if your required submission passes all tests.

Bonus submission on ADB, deadline: 26 Oct 2020, 23:00.