Parallel genetic algorithm and parallel simulated annealing algorithm for the closest string problem

This item was submitted to Loughborough University's Institutional Repository by the/an author.


Additional Information:

- This is a conference paper

Metadata Record: https://dspace.lboro.ac.uk/2134/2387

Publisher: © Springer Verlag

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
Parallel Genetic Algorithm and Parallel Simulated Annealing Algorithm for the Closest String Problem

Xuan Liu, Hongmei He, Ondrej Sýkora

Department of Computer Science, Loughborough University, Loughborough, Leicestershire LE11 3TU The United Kingdom
{x.liu, h.he, o.sykora}@lboro.ac.uk

Abstract. In this paper, we design genetic algorithm and simulated annealing algorithm and their parallel versions to solve the Closest String problem. Our implementation and experiments show usefulness of the parallel GA and SA algorithms.

1 Introduction

With the development of biology and pharmacy industry, some problems of bioinformatics become of great importance. One of such problems is finding the closest string representing a set of genes. Also creation of a drug, which would kill several closely related pathogenic bacteria, while it might be relatively harmless to humans, requires solution of the problem. Another example can be finding a consensus sequence, which is a single sequence that best represents a collection of related sequences.

Let us define the Hamming distance \(d(x,y)\) between two strings \(x\) and \(y\) of the same size as the number of positions in which \(x\) and \(y\) differ. We define the Closest String Problem as follows.

Closest String Problem (CSP): Given a set \(S = \{s_1, s_2, ..., s_n\}\) of \(n\) strings each of length \(m\), over an alphabet \(A\), find a string \(x\) of length \(m\) over \(A\) minimising \(d\) such that for each string \(s_i\) in \(S\), \(d(s_i, x) \leq d\).

As the CSP is NP-HARD [1], several different heuristic and approximation algorithms have been implemented. Lanctot et al. [2] designed \(\frac{3}{4}\) approximation algorithms, Li et al. [3] presented a PTAS. They used the standard linear programming and random rounding technique in their approximation algorithms.

In this paper we use machine learning approaches genetic and simulated annealing algorithms to solve the CSP.

The idea of genetic algorithm (GA) was originated by Holland in the 1960s. GAs are based on the principles of natural selection and adaptation and are claimed to be able to explore good solutions quickly in a large and complicated search space. The power of the algorithms comes from the mechanism of
evolution, which allows searching through a huge number of possibilities for solutions. It is that chromosomes are the information carriers and that the evolution process works at the chromosome level through reproduction. The reproduction can be made by either combining chromosomes from the parents to produce offspring, a process called crossover, or by a random change occurring in the chromosome pattern, termed mutation.

A GA creates an initial population of solutions at beginning. Then, the GA evaluates fitness function to all individuals of population, to characterise them from the most fit to the least fit. Afterwards, genetic operators transform the parent chromosomes to their offspring according to the criteria of fitness. The GA repeats the processes of selection, crossover and mutation to artificially simulate genetic operations. If the GA reaches the termination of the algorithm, the GA will output the best solution.

Simulated Annealing (SA) is an advanced Local Search method, which finds its inspiration from the physical annealing process studied in statistical mechanics [4]. The SA algorithm repeats an iterative procedure that looks for the better configurations while offering the possibility of accepting worse configurations. The SA algorithm provides opportunities to jump out from local optima.

Parallel computing has been a useful tool for improving running time and enlarge feasible size of problems with low cost. In this paper, we focus on the implementation of sequential and parallel GA and SA. We compared their performance by using different algorithm parameters and realised series of experiments focusing on the parallel processing issue.

2 Description of Algorithms

2.1 The Sequential GA for CSP

The GA generates an initial population \( P(t) \) of random candidate solutions \( ind_0, \ldots, ind_{\text{popsize} - 1} \) for a set of input string blocks to the CSP problem. Each individual of the initial population contains a string, (of length \( m \)) over the alphabet \( A \). Let \( d_{\text{max}} \) be the largest value of Hamming distance between an individual of the population and any string in \( S \).

We define the Hamming distance between two strings as the number of positions on which they differ. For example, given \( \text{string}_A = "\text{ACTGATTTGGCC}" \), \( \text{string}_B = "\text{GCTAGGTTCCGG}" \), the Hamming distance is 8, since there are 8 positions, on which the characters of the strings \( \text{string}_A \) and \( \text{string}_B \) are different. The fitness function is defined as the difference \( m - d_{\text{max}} \). A larger fitness value means a closer string, so we try to maximise the fitness value.

We use multi-point-crossover (MPX) in our GA as follows: two parental individuals, \( ind_x \) and \( ind_y \), are chosen randomly depending on the probability \( p_{\text{ind}} \). The probability of an individual is in proportion to the fitness of the individual for selection.

Two chosen parents exchange parts between two randomly picked points in the strings to form two offsprings. The two offsprings have a new order of string blocks, one part from \( \text{father} \) and the other part from \( \text{mother} \).
Afterwards, a mutation on any individual is executed with some probability $p_m$. Two positions are randomly chosen and exchanged in the individual. Repeat this procedure until the termination criterion is met. (see Algorithm 1)

**Algorithm 1** [Sequential Genetic Algorithm Structure]

1: $t ← 0$
2: initialize $P(t) = \{ind_i ∈ P(t), i = 0, 1, \ldots \text{popSize} - 1\}$
3: evaluate $P(t)$ to get the fitness of each individual
4: calculate the probability of each individual, $p_i ∝ ind_i.fitness$
5: $currBest = best\_ind(P(t))$
6: $bestInd = ind_{currBest}$
7: while ($t < \text{TERMINATION\\_CRITERION}$) do
8:  $i = 0$
9:  while ($i < \text{popSize}/2$) do
10:     select ($ind_x, ind_y$) from $P(t)$ according to their $p_{ind}$ - if two random values are located in the probability, $p_x$ and $p_y$ respectively
11:     \{chd_{2i}, chd_{2i+1}\} = crossover( ind_x, ind_y )
12:  end while
13:  $i = 0$
14:  while ($i < \text{popSize}$) do
15:      $r = \text{rand()} mod 100$
16:      if $r < p_m$ then
17:          mutate( chd_i )
18:      end if
19:  $P(t+1) ← P(t+1) ∪ chd_i$
20: end while
21: evaluate $P(t+1)$ to get the fitness of each individual
22: calculate the probability of each individual, $p_i ∝ ind_i.fitness$
23: $worst = worst\_ind(P(t+1))$
24: $ind_{worst} ← bestInd$
25: $currBest = best\_ind(P(t+1))$
26: if ($ind_{currBest}.fitness > bestInd.fitness$) then
27:      bestInd = ind_{currBest}
28: end if
29: $t ← t + 1$
30: end while

The algorithm terminates when the number of generations reaches a preset value.

### 2.2 The Parallel Island GA Approach

In this paper, we use island model to implement our parallel GA-CSP algorithm. The idea of Island Model is to distribute the total population to the available processors. A sequential GA runs with a sub-population on each processor. The sub-populations are independent from each other and therefore each processor
starts with a randomly generated subpopulation. Each processor will send its current best individual to another processor (randomly chosen with a probability, here, we set 1%) and receive the best individual from it in turn, and both replace the current worst individual by the received one.

2.3 The Sequential SA for CSP

Simulated Annealing is a generalisation of the Monte Carlo method for examining the equations of a state and the frozen state of $n$-body systems [5]. The idea comes from freezing of liquids or crystalisation of metals by the process of annealing. In this process, the system initially starts with a high temperature, then it is slowly cooled down until it approaches a "frozen" ground state. During the cooling process, the system is approximately in thermodynamic equilibrium. In the original Metropolis scheme [5] an initial state of a thermodynamic system was chosen at an energy $E$ and a temperature $T$, holding $T$ constant, while the initial configuration is perturbed and the change in energy $\Delta E$ is computed. If the change in energy is negative, the new configuration is accepted. If the change in energy is positive, it is accepted with a probability given by the Boltzmann factor $e^{-\frac{\Delta E}{T}}$. The system will iterate this procedure several times to get good sampling statistics for the current temperature, and then the temperature is decremented and the entire process is repeated until a frozen state is achieved at $T = 0$.

Simulated annealing has been used in various combinatorial optimization problems [4].

CSP belongs to a class of discrete minimisation problems, so we need to map the state of the thermodynamic system, which is analogous, to the current solution of the discrete problem as below.

- **Configuration**: A string over the alphabet, $\{A\}$, which has the same length as the strings in $S$.

- **Rearrangement**: A point in a string is picked randomly, then the two parts divided by the point are exchanged.

- **Energy Function**: $E$ is defined as follows: $E = \min_{s_i \in S} d(s_i, x)$, where $x$ is the current solution string.

- **Annealing Schedule**: Avoidance of entrainment in local minima is dependent on the "annealing schedule", the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds. This requires a lot of experimentation. We first do some random rearrangements, and use the results to determine the range of values of $\Delta E$ that will be encountered from one state to another state. Choosing a starting value for the parameter $T$ which is considerably larger than the largest $\Delta E$ normally encountered. In our experiments, $T$ is half of the possible maximal Hamming Distance, which is the length of string, so $T = m/2$. At each temperature 100 iterations are
performed. The temperature reduction factor, $\gamma$, is set to 0.9. When the termination criterion, $T = 0.001$, is met, algorithm stops. (Algorithm 2)

**Algorithm 2** [Sequential SA algorithm for CSP]

1: randomly generate an initial string $s_c$
2: set an initial $T = T_{max}$
3: set an initial repeat times $L$
4: set $\gamma$

5: while termination = false do
6: for $0 \leq I < L$ do
7: randomly choose string $s_n$
8: $\Delta = f(s_n) - f(s_c)$
9: if $\Delta \leq 0$ or ($\Delta > 0$ and ($e^{-\frac{\Delta}{T}}$ is verified)) then
10: $s_c \leftarrow s_n$
11: end if
12: if $T < T_{min}$ then
13: termination = true
14: end if
15: end for
16: $T \leftarrow \gamma T$
17: end while
18: output

2.4 The Parallel SA Approach

We used PVM to implement our parallel SA algorithm for CSP. The main idea of parallelism comes from the genetic algorithm Island Model. Let each processor run the sequential SA algorithm independently. At a randomly chosen time, the master in PVM randomly chooses two processors, and the current temperatures of these processors are exchanged. Each of the two processors will set the received temperature to be its initial temperature for the next annealing process. Each processor will return its best result to the master after having reached the termination-condition. Otherwise, the processors will carry on in the annealing process and exchange their temperature with others on the master request.

3 Test Results

We have two kinds of test platform, one is that we fix the string blocks number and change the string length (see Table 1), another is that we fix the string length and change the string blocks number (see Table 2). For each of the randomly generated problem instances every algorithm was run 5 times. We discarded the worst results and got the average results. We used the following configurations of
sequential GA: the total population size was 50, the probability of mutation was 0.5 and the number of generation was 2000. For our sequential SA algorithm we used the following configurations: the initial temperature, $m/2$, the temperature reduction factor 0.9, and the minimum temperature 0.001.

We tested our PVM implementation of the algorithms on a cluster of 20 Sun ULTRAsparc workstations running Debian GNU/Linux. They are connected with 100Mbit Ethernet using Cisco 2950 switches. We compared the sequential and parallel GA and SA algorithms. Both parallel algorithms used 4 processors. In our parallel Island Model GA, we set only 1% opportunity that any two processors will exchange their individuals.

<table>
<thead>
<tr>
<th>String Size</th>
<th>GA</th>
<th>SA</th>
<th>Parallel-GA</th>
<th>Parallel-SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>9</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>40</td>
<td>13</td>
<td>12</td>
<td>17</td>
<td>14</td>
</tr>
</tbody>
</table>

**Table 1.** The average test results for 10 strings and different string size. The results in the figure are the total length of string minus the maximal values of the Hamming distance between the return string and each of strings in $S$.

From our results, one can see that the parallel GA produces better results than others. The parallel algorithms produce better results than their sequential versions. One possible reason is that parallel algorithms efficiently use the processors to search in a larger solution space. Our tests show usefulness of parallelism: increasing the number of processors in the parallel GA and SA algorithms, produces better solutions.

<table>
<thead>
<tr>
<th>Strings</th>
<th>GA</th>
<th>SA</th>
<th>Parallel-GA</th>
<th>Parallel-SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>30</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 2.** The average test results for the strings, which are 20 bits long, but with different string block numbers.

### 4 Conclusions

In this paper, we designed sequential GA and SA algorithms for CSP. Comparison of all four parallel and sequential algorithms shows the superiority of the parallel island GA algorithm and usefulness of the parallel versions of our machine learning algorithms.
References