Evolutionary Algorithm for Optimizing Parameters of GPGPU-based Image Segmentation

Sándor Szénási[1], Zoltán Vámossy[2]

[1] Óbuda University, Bécsi 96/B, H-1034 Budapest, Hungary, szenasi.sandor@nik.uni-obuda.hu

[2] Óbuda University, Bécsi 96/B, H-1034 Budapest, Hungary, vamossy.zoltan@nik.uni-obuda.hu

Abstract: The use of digital microscopy allows diagnosis through automated quantitative and qualitative analysis of the digital images. Often to evaluate the samples, the first step is determining the number and location of cell nuclei. For this purpose, we have developed a GPGPU based data-parallel region growing algorithm that is equally as accurate as the already existing sequential versions, but its speed is two or three times faster (implementing in CUDA environment), but this algorithm is very sensitive to the appropriate setting of different parameters. Due to the large number of parameters and due to the big set of possible values setting those parameters manually is a quite hard task, so we have developed a genetic algorithm to optimize these values. Our evolution-based algorithm that is described in this paper was used to successfully determine a set of parameters that compared to the results with the previously known best set of parameters means a significantly improvement.

Keywords: biomedical image processing, nuclei detection, GPGPU, CUDA, genetic algorithm

1 Introduction

Our work focuses on the segmentation of images containing hematoxilin-eosin stained colon tissue samples (Fig. 1). There are several procedures to identify the main structures in these images and a lot of them based on a reliable cell nuclei detection method (these procedures need the exact locations of the cells).

There are several image processing algorithms for this purpose [1][2][3][4][5], but some factors could increase the challenge. The size of the images can easily reach the order of few 100 Megabytes, therefore the image processing speed plays an important factor.

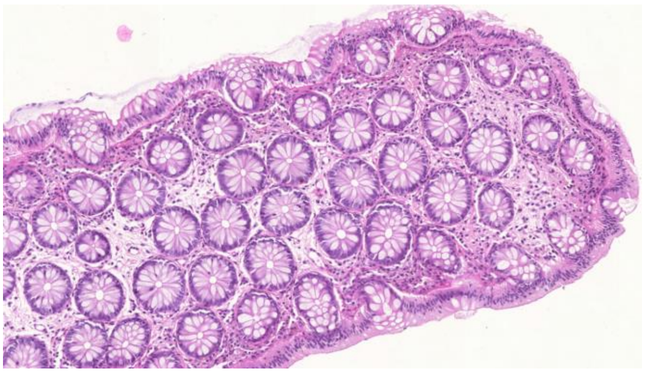


Figure 1
HE stained colon tissue

One of the most promising alternatives is the region growing approach; it can correctly separate structures but it is too slow for practical usage. Parallelizing the region growing algorithm aims at providing better execution times, while delivering the similar outcome produced by the sequential version [4]. We have developed a GPGPU based region growing algorithm [6], and implemented it in CUDA environment. The new method is 25–65% faster than the CPU based implementations and its accuracy is the same.

The different steps of the region growing algorithm (pre-processing, the region growing itself, separation of merged cells) require fairly much (in our implementation, twenty-seven) parameters which can be considered as independent variables since their effects on each other is unknown. Every parameter greatly affects the output of the algorithm, so to define an optimal set of parameters we have to treat all parameters simultaneously. Due to the large number of the parameters manual method is practically hopeless; we need an intelligent system [7] that finds the best values. Therefore we have developed a genetic algorithm, which tries to find the most optimum set of parameters from the available parameter collection.

2 Description of the Parallel Region Growing Algorithm

2.1 Detection of Seed Points

In the first step of region growing, we have to find the most intensive point of the image that complies with some. In case of the GPGPU implementation this means multiple points, becase we can execute multiple cell nucleus searches parallel. The adjacent seed points can cause problems (we must avoid overlapping cell nuclei), which would require a lot of computational time to administer. We know the maximum radius of a cell nucleus, so we can presume that the searches started from two seed points (that are at least four times further apart than this known distance) can be handled as independent searches; so they can be launched in the same time.

A quick overview of our complex searching algorithm (full description about this algorithm is given in our previous paper [6]):

1. The points that matches the starting condition and that have the biggest intensity must be collected into an $S_{waiting}$ set (since we only store the intensity on 8 bits, it is likely that there will be more than one points). To achive this we can use the atomic operations of the GPGPU.
2. One element is selected from the $S_{waiting}$ set, and it is moved to the $S_{confirmed}$ set. We can use the GPGPU capabilities therefore every seed point is checked by a seperate thread.
3. In the $S_{waiting}$ set, we examine the next element: we check if any of the elements from the set $S_{confirmed}$ collide with the parallelized processing of this element (they collide, if the distance of the two points is below the critical threshold). If there is no collision, then this element is moved into the $S_{confirmed}$ as well, otherwise it stays where it is. We repeat step 3 until we run out of elements in the $S_{waiting}$ set, or we find no more suitable points, or the $S_{confirmed}$ set is full (its size is the same as the number of the parallelized region growing runs we want to execute simultaneously).
4. We launch the region growing kernel using the seed points that are in the $S_{confirmed}$ set.
5. After the execution of the kernel, we store the results, we delete the contents of the $S_{confirmed}$ set, and the elements from the $S_{waiting}$ set that no longer match the starting criteria.
6. If there are still elements left in the $S_{waiting}$ set, we continue with step 2. If it is empty, we continue the processing with step 1.

This iteration is continued until the thread runs out of seed points, or the required amount of points is enough for the starting of the next region growing.

2.2 Cell Detection with Parallel Region Growing

The region growing itself iterates three steps until one of the stop conditions is met. A quick overview of the region growing iterations (full description about this algorithm is in our previous paper [6]):

1. We have to check all the possible directions in which the contour can be expanded. This means the four-neighborhood of the starting point (in case of first iteration) or the pixels around the lastly accepted contour point (in case of further iterations). The examinations of these pixels are independent of each other, therefore we can process them in the same time: four threads

examine the different neighbors, whether they are suitable points for further expansion or not.

2. All of the contour points must be evaluated to decide in which direction the known region should be expanded. For this, we have to evaluate a cost function [4] for every point. It is important to notice that some parameters of the cost function change at the insertion of every new point, so they have to be re-calculated in every iteration for every points. This is well parallelizable calculation, every thread counts the cost of a single contour point.
3. The contour point with the smallest cost must be selected. We can use the *atomicMin* function to make the threads calculate the smallest cost. That contour point can be chosen to increase the region.

After every iteration, a fitness function is evaluated that reflects the intensity differences between the region's inner and outer contour, and the region's circularity. The process goes until the region reaches the maximum size (in pixels or in radius), and its result is the state where the maximum fitness was reached [6].

We can start several region growing parallel. In this case we have to assign a single block to the processing of one single cell nucleus.

3 Parameter Optimization Techniques

It can be reasonably simple to create a model of the problem itself: we have 27 mutually independent parameters with pre-defined target sets; we have a working region growing algorithm (and its implementation) that produces the list of cell nuclei (size, location, etc.) that can be found in a sample; and we also have an evaluation function [8] that determines an accuracy value that is defined by comparing the results of the aforementioned region growing algorithm with the slides that were annotated manually by qualified pathologists (further on, the "Gold Standard" or GS slides) available at our disposition. Our aim is to search a set of parameter values that gives us the best possible accuracy (which is a classical problem in the field of image segmentation [9]). This is a classic optimization task and as for such, we found several possible solution alternatives.

3.1 Enumerative Methods

The different search methods can be grouped into three distinct categories [10]: random, enumerative and calculus-based methods. The basic principle of the enumerative-based methods is to examine all possible solutions and then it selects the best from those [11]. This method could easily be adapted to our task; we should only list all possible combination of parameters, and then we could find the best by using a simple linear search (thought it would be a good question that what resolution should be used to break up the different intervals into discrete values).

It is guaranteed that this solution finds the best solution, but due to the large number of possible combinations, this method is practically useless for us. Thought we can influence the number of possibilities at some level (by arbitrarily narrowing down the target sets or decreasing the resolutions of the intervals); but in the case of a typical real-world configuration the evaluation could probably take years. And since we do not have exact information about the results of the various parameters and about their effects on each other, we cannot even reduce the search space using different heuristic methods either.

3.2 Calculus-based Methods

The other important group of methods contains the calculus-based methods where the reasonably large search space does not necessarily mean large computational needs, because there is no need to search through the whole problem space. In these cases, we start off an arbitrarily selected point, and then the algorithm (after every single evaluation) continues in the direction that at the given moment results in the greatest immediate positive advantage (the classical examples of this approach are the Greedy [12][13] and the Hill-Climbing [14] methods).

Naturally we can implement this method to our problem as well: as a starting point, we can select any point in the parameter space, and then we can evaluate its immediate surroundings, and then we can move towards the point that has the biggest accuracy and continue our search from that place. These greedy algorithms can be reasonably fast in finding a result, but we can only guarantee that this result of parameter set will be a local optimum; however, we will have no clue about the relation of this local optimum with the global optimum. Since the parameters are loosely connected to each other, it is very likely that we will find many local maxima (points where the parameter set yields us reasonably good results but a slight modification will worsen these results).

The disadvantage of this method is that there is no possibility for a backtrack, so once the search has ran into such a local maximum, we can only get to a different result by re-starting the whole process from a different starting point.

The search is sensitive to its starting parameters, so of course it is possible to fine-tune the results if we simply start several different search processes from different starting points, and then we compare the results of these. However, due to the large number of parameters, there can be many different possible starting points, and within a finite time, we can only examine a very small fraction of these. Furthermore, the large number of parameters poses another problem: because of this, the examination of the neighborhood of a single point can have very high computational needs if we want to test the outcome of all possible directions (what is more, due to the large number of parameters, we would be forced to simplify this process, so we would lose the biggest advantage of the hill-climbing algorithm: that is, we risk that it will not find the local maximum as it originally should).

3.3 Guided Random Search Techniques

The third group of search methods is the random search group: in a strict sense it simply means that we examine randomly selected points within the problem space and we always store the one that yields the best results. On its own it will be less effective for us, for two reasons. Firstly, its execution time can get pretty high. Secondly, it gives us absolutely no guarantee of finding any kind of local or global optima within finite time. However, in practice, we can use the guided random search techniques [15]: they basically examine randomly selected points, but they try to fine-tune the selection method using various heuristics.

Typical examples of these methods are the Tabu Search [16], the Simulated Annealing [17] and the Evolutionary Algorithms [15]. Out of these, the different evolution-based algorithms are the most interesting for us. The basis of these techniques is that various well-known biological mechanisms are used to execute various search and optimization algorithms. These algorithms share the same basic principle: newer and newer generations are calculated starting from an initial population (in our case, this means a starting set of parameters for the region growing algorithm); and the members of the new generations are expected to be more and more viable (in our case, this will represent parameter sets that yield more and more accurate results). In addition, we can define various stopping criteria as well, but this method can also be used even without those, because the individual generations can be checked on-the-fly, so the parameter set that actually yields the best results can always be accessed at any given moment.

In our case, this technique seems the most suitable, because our task fulfills the following criteria where the usage of the genetic algorithm is considered favorable:

- Search space with multiple dimensions, where the relations between the major variables are unknown.
- The traditional solutions give us unacceptable execution times.
- It is very difficult (or impossible) to narrow down the search space.
- One solution can be checked quickly, but finding an optimal solution is difficult.
- We do not necessarily need the global optimum, we merely want the best possible result.

```
Formulate the initial population()
Randomly initialize the population()
Repeat
   Evaluate the objective function()
   Find fitness function()
   Apply genetic operators (reproduction, crossover, mutation)
Until (¬Stopping_critera())
```

Algorithm 1
Typical structure of genetic algorithms

In addition to this, the genetic algorithms have several more advantages:

- They can be very well parallelized so they can be efficiently implemented in multi-processor environments. Since the method described above has reasonably big computational requirements (because of the fact that during the search process both the region growing itself and the evaluation too have to be executed for every set of parameters); it is essential that we can complete this process within the shortest possible time in a distributed environment.
- Compared to the hill-climbing method, it is more likely that we find a global maximum because due to the nature of its operation a genetic algorithm continues the search even if a local maximum is found: due to the mutations, there are nonstop changes even with a stable population. In addition to this, even thought we cannot guarantee that it will find the global maximum within a finite time, but there are several papers that show that the search itself generally converges towards an optimal result [18], [19], [20].
- After reviewing the publications in this topic we examined several genetic algorithms for optimizing parameters and usually these algorithms met the previously stated requirements [21], [22].

4 Construction of the Genetic Algorithm

4.1 Representation of Chromosomes

A genetic algorithm usually contains the steps of Algorithm 1. When creating a genetic algorithm, a decision must be made that which encoding form should be used with the data of the different instances (in our case, an instance is represented by a single chromosome so we use the two terms as equals). For this, the search space must be somehow mapped to the chromosome space. To do that, we have several possibilities, but the following aspects must be taken into consideration:

- We want to optimize a reasonably big set of parameters.

- These parameters can be considered as being independent from each other.
- Every parameter is a number; some of them are floating point numbers.
- The target sets of the different parameters are different.
- Most of the parameters can be surrounded with an upper and lower bound, but there are several parameters that have to comply with additional rules as well (e.g. only even numbers are allowed).

It is a very common approach that the parameter set is simply converted into a vector of bits (using the values of the parameters in base-two form). However, several researchers suggest [23] that we should not encode the floating point numbers in bits; the chromosomes should contain real (floating point) numbers. The advantage of this method is that we can easily apply problem-specific mutations and crossovers.

In our case we have a multi-dimensional search space; and according to this we have to store several genes inside a chromosome (these genes actually represent the parameters of the region growing algorithm). Since the relations between the individual genes are hard to describe, it is more feasible to choose a representation where every parameter is separately encoded according to its target set and where we can establish that the functionality of the different genes do not depend on their location inside the chromosome.

Of course, we can have chromosomes that are not viable (if not the crossovers, then the mutations will probably produce some of those). These instances are handled in a way so that they will have a fitness value of zero. This way (even thought they will be indeed evaluated) these chromosomes will most certainly not be included in the next generations.

4.2 Initial Generation for Genetic Algorithm

The initial generation is generated by randomly generated instances. There is a set of parameters that is already used in the live applications, but using this set as the initial generation came with no benefits at all, because the system found an even better set of parameters within only a few generations).

Some of the parameters have known upper and lower bounds [24]. In case of these parameters the actual values for the different instances were chosen using Gaussian distribution within these intervals. The known intervals are:

- Cell nuclei size: 34 – 882 pixels
- Cell nuclei radius: 4 – 23 pixels
- Cell nuclei circularity: 27.66 – 97.1
- Cell nuclei average intensity: 36.59 – 205.01 (RGB average)
- Seed point intensity: 0 – 251 (RGB average)

With some technical parameters it is not possible to perform such preliminary tests, in these cases the initial values of the parameters are distributed using the currently known best set of parameters. The intervals are generated in the ±10% surroundings of these aforementioned values using normal distribution. Of course, this does not guarantee that the optimal result will be within this interval, so it is advised to check both the final result of the optimization and the intermediate results as well: if the instances that yield good results have a parameter that is always around one of the arbitrarily chosen bounds, then it might be practical to further extend this interval. Despite this (due to the operation of the genetic algorithm, and due to the mutations) the genes can have values that are outside these starting intervals, so an initial generation generated with some unfortunate initial values can also find the good final result as well.

Every instance in the first generation was created using random parameter values, so with a great probability there are a large number of non-viable instances in this generation, which is not feasible for the further processing. Firstly, it is necessary that we have the biggest number of usable instances for the next generations so that we can choose the best candidates from multiple possible instances. Secondly, there is a bigger importance on the first generation, because it would be better if there could be the biggest number of different values for a parameter so that these multiple values can be taken into the next generations and this way there is a bigger chance that the given parameter’s optimal value is carried along as well. For these reasons, the first generation is created with a lot more instances than the following ones, in our case that means 3000 chromosomes. The following generations will contain only 300 instances (both values were determined arbitrarily based on the experiences of the first test runs).

4.3 Processing Generations

We have to calculate the fitness value of all instances in the actual generation. In our case the fitness value is determined based on how well the parameters represented by the chromosome performs with the region growing algorithm. We need two main steps to calculate this:

1. First we have to execute the region growing cell nuclei detector algorithm with the given set of parameters. Since the parameters can behave differently on different slides, the region growing is executed for several slides.
2. The next step is the evaluation of the result of the region growing (the list of cell nuclei). For this, we compare these results (test result) with the manual annotations of the Gold Standard slides (reference result). By averaging the results of the comparisons we get the fitness value for the given chromosome.

4.3.1 Accuracy

We use the evaluation algorithm described in our previous paper [8] for comparing test and reference cell nuclei sets. This is based on the confusion matrix [25] that can be constructed using comparison the two result sets hits (another approach can be a fuzzy based model [26], [27]). The accuracy of the region growing is a simply calculated measurement number [25]:

$$Accuracy = (TP + TN) / (TP + TN + FP + FN) \qquad (1)$$

Where TP means the number of *true-positive* pixels; TN is the number of *true-negative* pixels; FP is the number of *false-positive* pixels and FN means the number of *false-negative* pixels.

The pixel-level evaluation itself will not give us a generally acceptable result, because it will only show a small error even for big changes in densities, and our task is not only to determine if a pixel belongs to any object or not, but we have to locate the objects themselves (because it is not acceptable to detect one big nucleus instead of a lot of small nuclei).

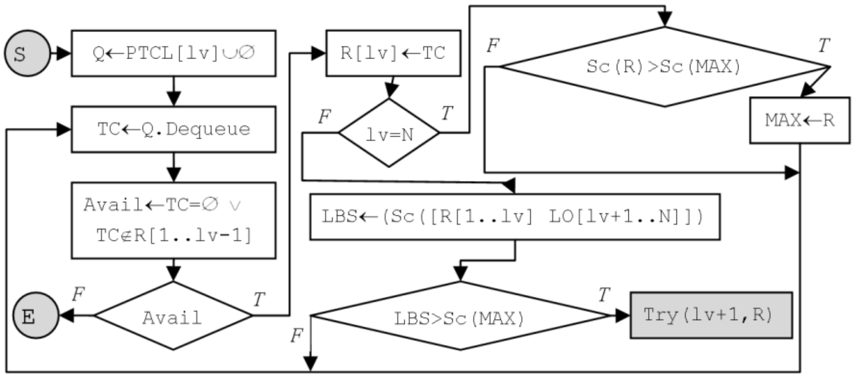
4.3.2 Object Level Evaluation

Our measurement number does not based only on a pixel-by-pixel comparison; instead it starts by matching the cell nuclei together in the reference and the test results. One cell nucleus from the reference result set can only have one matching cell nucleus in the test result set and vice versa. After the cell nuclei matching, we can compare the paired elements.

The final result is highly depends on that how the cell nuclei are matched against each other in the reference and the test result sets. Due to the overlapping nuclei, this pairing can be done in several ways, therefore it is important that from the several possible pair combinations we have to use the optimal (the highest final score).

During the practical analysis [8] of the results we found out that on areas where cells are located very densely, we have to loop through a very long chain of overlapped cells, which results in groups that contain very much cell nuclei from the test and from the reference sets as well. We use a new backtracking search algorithm [28] to find the optimal pairings. This requires significantly fewer steps than the traditional linear search method.

The sub-problem of the backtracking search is finding one of the overlapping cell nuclei from the test results and assigning it to one of the reference cell nuclei. It is possible that there will be some reference cells with no test cell nucleus assigned to them and it is also possible that there will be some test cells with no assignment at all. The final result of the search is the optimal pairing of all the possible solutions. Full description of this algorithm is given in our previous paper [8].



Algorithm 2

Backtracking core algorithm

Inputs and utilized functions:

- lv – The level currently being processed by the backtracking search.
- R – The array that holds the results.
- PTCL – An array for every reference cell with test cells from the group that overlap with the given reference cell.
- LO – For every reference cell nucleus LO represents the local optimal result, if we choose the best overlapping test cell. During the search, the algorithm will not move to the next level if it is found out that the final result will be worse if we choose the most optimal choices on all the following levels.
- Sc(X) – It returns the value for the pixel-level comparison using input X (which is a pair of a test and a reference cell nucleus).

Algorithm 2 will search and return the optimal pairing of a group containing test and reference cell nuclei. The i^{th} element of the MAX array shows that the i^{th} reference cell nucleus should be paired with the MAX[*i*] test cell nucleus.

The above algorithm algorithm should be execued for every group to get the optimally paired cells. After that, we can calculate the above mentioned accuracy. This will be the fitness value of the given chromosome.

4.4 Implementation of Genetic Operators

4.4.1 Implementing Selection Operator

The selection of the parent pairs can be performed using any available methods; the important feature is that instances with a bigger fitness value should be selected proportionally more times than the others. One of the best solutions for this is the well-known roulette wheel selection method, where we generate an imaginary roulette wheel where every instance has a slot that is sized proportionally according to the fitness value of the given instance. When selecting the new parents, we spin the wheel and we watch which slot the ball is falling into [29].

The probabilities can be determined using the following formula:

$$P_i = (F_i - \mathrm{Min}(F)) / \sum_k (F_k - \mathrm{Min}(F)) \qquad (2)$$

Where

- P_i: the probability to select the instance #i
- F_k: the fitness value for the instance #k
- Min(F): the smallest fitness value for the current generation

It is visible that by using this calculation method there is a side effect: the instances with the smallest fitness values are completely locked out from the next generation, but in the practical implementation this means absolutely no problem. However, this way we eliminate the biggest drawback of the roulette wheel method too that occurs if the fitness values for different chromosomes are too close. In addition, since it is possible that non-viable instances are in the generation as well (which means that after the evaluation there can be some chromosomes with the fitness value of zero) we have to remove these instances before calculating the probabilities so that these instances will not distort the minimum value in the equation.

The search space is reasonably large, so the occasionally occurring instances with good fitness value can disappear in the next generation due to the random crossovers. For this reason we use elitism [30]: the instances with the highest fitness values are carried along into the next generation (10% of the generation is selected this way). This method slightly decreases the number of trial runs per generation, but this way we guarantee that the best chromosomes are kept.

4.4.2 Implementing Crossover Operator

The most typical step of a genetic algorithm is the crossover. There are several known methods for this step; the simplest one is the single-point crossover: we simply cut the chromosomes in half using a random cut-point and we exchange the chromosome parts that are after the cut point [18]. This can naturally be done using two or more cut points as well, the most generic case being the uniform crossover where we generate a random crossover mask that simply defines for every bit which parent is used for the given bit of the descendant instance.

In our case the size of the population can be considered as reasonably small (because the evaluation of the single instances can be very time consuming, so we cannot use a large population); while the chromosomes are considered to be reasonably large (27 parameters, several hundred bits altogether). Because of these we clearly have to use the uniform crossover method.

In the strict approach, the uniform crossover could be performed for every bit; but in our case this is not feasible. The reason for this is that there are some parameters that have to comply with some additional rules (e.g. divisibility), and the bitwise mixture of those can easily lead us to values that do not belong to the target set.

We only combine whole genes: for every gene we use a random number to determine which parent's gene is inherited.

We use the following probabilities:

$$P_a = (F_a - \text{Min}(F)) / (F_a + F_b - 2*\text{Min}(F)) \quad (3)$$

$$P_b = (F_b - \text{Min}(F)) / (F_a + F_b - 2*\text{Min}(F)) \quad (4)$$

Where

- P_a: the probability of gene of parent A is inherited
- P_b: the probability of gene of parent B is inherited
- F_a: the fitness value for parent A
- F_b: the fitness value for parent B
- Min(F): the smallest fitness value for the current generation

4.4.3 Implementing Mutation Operator

The size of the mutation cannot be defined in a general form (for every parameter), because the values of the parameters can be very different. With some parameters small changes in the actual value can have great effects, while other parameters are much less sensitive. For this reason, we rejected the bit-level mutation, because in some cases this can result in too drastic changes.

Some parameters are reasonably small integers, and we cannot use percentage-based mutations with those, because neither the small nor the medium-sized mutations would be enough to change the value even to the closest integer. For this reason these values change with discrete values according to the $\pm\varepsilon$ method described in [31]: small mutation means a change with one, medium mutation means a change with two, and a big mutation means a change with three; and there is a 50–50% chance that the change will be positive or negative.

During the mutation process we do not check if the new value complies with the given parameter's target set. If a mutation causes the parameter to have a wrong value, then the chromosome will have a very bad fitness value (if the parameter is out of bounds, then a zero fitness value) after the next evaluation, so similarly to the rules of nature, these non-viable instances will fall out from the next generations. The number of such instances is quite small, so this does not deteriorate the results of the search.

By introducing an auxiliary verification step this could be resolved, but on one hand the development of an extra rule-set is difficult (since we do not know exactly the relations between the various parameters), on the other hand it would not be feasible to fix these rules. The reason behind this is that the mutations have an additional important role in the system: since in the starter generation we generated the parameter values between arbitrarily chosen interval boundaries, it is possible that due to a mistake in this process, the ideal gene was not even present in the initial generation. In this case, the mutation can help us so that the search continues to an undiscovered area that has great potentials and that was originally locked out due to a badly chosen interval (or an unlucky random number generation).

The probability of a mutation is 10%. The size of the mutation is a random number based on the parameter (generally there is a 60% chance for small, a 30% chance for medium-sized, and a 10% chance for large mutation).

5 Implementation Details

The actual implementation was done according to the following criteria:

- 27 parameter values are searched.
- The initial generation has 3000 chromosomes.
- Every following generation has 300 chromosomes.
- Every parameter set (every instance) is tested against 11 representative tissue samples.

With the genetic algorithms, usually the execution time is the only barrier for the accessibility of the optimal results. This is true in our case as well, because the evaluation of a single instance needs quite a lot of time: based on the first 1550318 evaluations, the average processing time for a single region growing is 1498ms, and it took another 8249ms to evaluate the results for a single image. Since we always have to examine 11 tissue samples, it takes about 107 seconds to examine a single chromosome, so the evaluation of a single generation takes about 8.93 hours.

Since we have to assume that we will process hundreds of generations, it is obvious that we have to speed up the whole process somehow; and an obvious method for this is a high-level parallelization. Out of these, the most basic method is the master-slave processing [29]. Since the testing of the single instances (the execution of the region growing and the evaluation of the results) can be considered as independent tasks, these can be efficiently parallelized in a distributed environment. In this case we work with a global population and the parallelization is merely used only for the evaluation. To achieve this, a parallel execution environment was developed that uses client-server technologies to execute the genetic algorithm using the parallelized method described above.

A single central server manages the whole process. Basically, this program stores the population itself and this is where the basic genetic operations are executed. The main tasks of this module are:

- Generate an initial population.
- Generate any following generations (selection of parents, crossover, and mutation).
- Distribute the tasks towards the evaluator clients, then collect and process the results.

The number of clients that can connect to the server is not limited; the clients do not see the whole population and they take no part in the genetic operations, they only perform auxiliary operations for the evaluation. Their tasks are:

- Download the data that are necessary to process an instance (tissue images, parameters).
- Execute the region growing algorithm for every tissue sample using the downloaded parameter set.
- Evaluate the results of the region growing algorithm, and then send back the results of the evaluation to the server.

The system was designed with scalability in mind, because the number of available client computers greatly varies over time. In the worst case this can be as low as only a few clients, but sometimes (by taking advantage of the university's capabilities and using the occasionally free computer laboratories) it peaked at a little more than one hundred. The server is continuously online and produces the data packets that require further processing; the clients can join at any time and can operate for any length of time. The smallest atomic unit is the processing of a single chromosome.

Although there is no technical upper limit for the number of connected clients, we usually used 100 clients, which proved to be enough for our generation size of 300 instances. Increasing the number of clients does cause an increase in speed, but this increase is far from linear. The cause behind this is that evaluating the different instances always requires a different amount of time, and in nearly every generation there are some clients that receive a task unit that requires a lot more computational time than the others. So by the time these clients are finished with their one instance, the other clients can process two or even three of them. So even if we increased the number of clients, the bottleneck would still be the single client that received the most difficult instance.

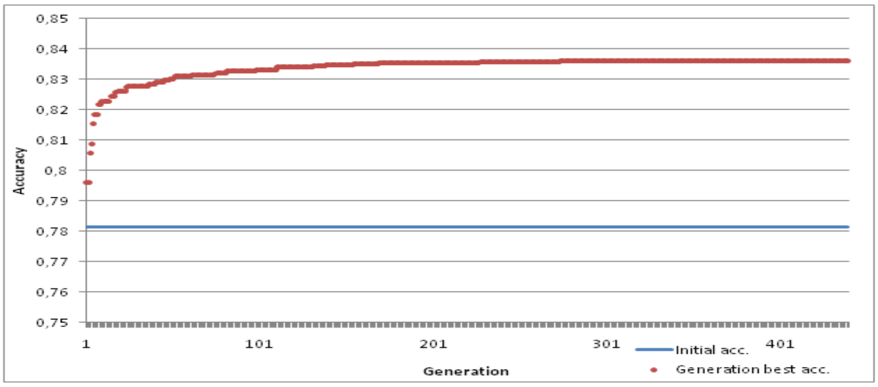


Diagram 1
Best accuracy by generations

The extensive description of our developed system (structure of the server and the client, description of the communication and the robust implementation) will be detailed in our next paper. In the followings, we present our results after 440 generations (this took approx. 3948 working hours; thanks to parallel processing capabilities this was only a weekend in practice).

6 Final Results

6.1 Examination of Generations

The best fitness values give us a monotonically increasing series of numbers because the chromosomes with the best fitness values are automatically carried along into the following generation as well. These values can contain useful information about the speed of our optimization.

We did not have any specific goals for this task, so we cannot mention any expected results that we wanted to get (with the exception of the 100% accuracy; although, reaching this seems impossible, because we have to consider that the region growing itself does not guarantee a perfect result, even if we find the optimal set of parameters). For these reasons, our aim is to get the best possible results, compared to our possibilities.

We can compare these values to the parameter set that was originally developed by Pannon University [4], which gave us the average accuracy of 78.1%. This can be considered as the acutally known best accuracy. Surpassing this value with any extent can be considered as a useful result.

Diagram 1 shows the best results for every generation, the horizontal line shows the 78.1% level. In the first generation we even had an instance that reached this level; and later we managed to even improve this accuracy. By generation #273, the best accuracy reached at 83.6%; this means an upgrade of 5.5% compared to the previous best result.

We kept the algorithm running for even more time, but until we stopped it (at generation #440) it produced no generation that had better accuracy than this level.

6.2 Examination of the Stopping Condition

In the case of genetic algorithms it is usually very hard to determine an exact stopping condition. Our task is a great example for this as well, because we do not have a specific pre-defined final goal (final accuracy) that we wish to reach. It can be practical to examine the differences between the individuals of the consecutive generations; expecting that those will be more and more similar to each other, so by examining the deviation in the accuracy of the parameter sets represented by the various chromosomes, we could stop the search if it goes below a pre-defined limit. However, in our case this cannot be used, due to the fact that we use relatively big mutations. For this reason, we examined the parameters separately, and by checking the changes in their values, we made the decision whether to stop the algorithm or not.

Due to lack of space we cannot give extensive details about all the 27 parameters, so we only describe some more interesting specific examples. The changes of the parameters over time are shown on a similar to the previous ones: horizontal axis represents the generations, vertical axis represents the values of a single parameter in the chromosomes of the generations using grey dots, so in this case as well, the darker areas obviously mean overlapping values (all referenced diagrams are available on the following website: *http://users.nik.uni-obuda.hu/sanyo/acta835*). It can be stated that we noticed three distinct patterns in the changes of the various parameters:

Most of the parameters have settled to their respective ideal values reasonably fast (usually this happened around generation #100), and then their values did not change from this level. Obviously due to the mutations there were some values below or above this ideal level, but all of these turned out to exist only for a short period of time. Good examples for this pattern are parameters #1 and #2 (Diagram A.1 and A.2), and even thought the required time for the stabilization is a little higher, the same pattern is visible with parameters #3 and #6 (Diagram A.3 and A.4).

It is less common, but it does happen with some parameters that can also be known as a “typical example” of genetic algorithms, which is when different parameter values (alleles) compete with each other. In these cases, a few different parameter values remain viable for a reasonably long period of time, sometimes even for 40–50 generations. During this time, some of them can get stronger (sometimes even two or three different values can remain dominant simultaneously). But eventually, some stable state was achieved with these parameters as well; good examples for this pattern are the parameters #13, #19 and #24 (Diagram A.5, A.6 and A.7).

In addition, we can find some examples for genes whose values did not settle, not even in the last generation (#400) when we stopped the algorithm. Examples for this pattern are the parameters #9 and #12 (Diagram A.8 and A.9).

It is visible that in these cases there were usually only two values in the end, and these cases are usually next to each other, so it basically does not matter which one we choose. For this reason, we should not continue the search just for the sake of these parameters.

6.3 Examination of the Number of Non-Viable Instances

Although this factor is less important for us, but it is practical to examine the number of non-viable instances inside the different generations.

During the creation of the first generation and later during the execution of the crossovers and the mutations we performed no verification whether or not the newly created instances' parameters are valid or not. This is because we had the assumption that the newly created non-viable instances will fall out at the next selection for parents, so hopefully there will be less and less non-viable chromosomes as the generations evolve.

Diagram 2 seems to verify this assumption: the first (initially created) generation's data is not shown, because there were 1879 (from 3000) non-viable instances in that generation. This number drastically decreased in the following 5 generations, and then it was stabilized around 13.2 items.

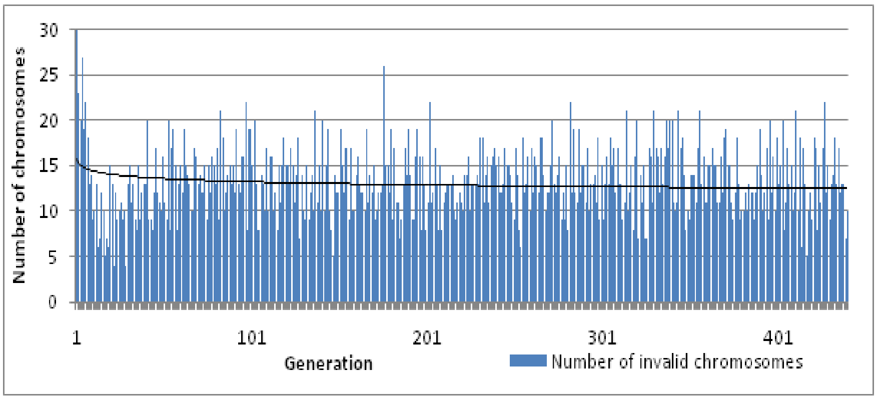


Diagram 2
Number of invalid chromosomes

Since the crossovers and especially the mutations can still create non-viable instances in the subsequent generations as well, we cannot expect that this number will furthermore decrease; but this is a tolerable amount, and the evaluation of those instances does not require too much resources.

6.4 Examination of the Control Group

Since the region growing and the evaluation have both high computational needs, we have to decrease the number of necessary processing steps as much as possible. One possible way of doing so is that we do not run the evaluation for every possible tissue samples (which would be altogether 41); instead we selected 11 tissue samples so that they reflect all different kinds of sample images and there is at least one example for every sample type (e.g. a lot of cell nuclei or only a few; sharp or blurred contours).

We executed the search only for this narrowed list, which (in addition to the faster evaluation of the instances) has the additional benefit that we could use the remaining sample images as control group. According to this, by executing the cell nuclei search algorithm for all possible samples, we got the following results:

- Average accuracy using the old (best known) set of parameters: 76.83%
- Average accuracy using the parameter set found by our evolution-based algorithm: 81.15%

Since these samples were not used during the optimization process, we can very well use them to estimate the improvement of accuracy with unknown samples, and in our case this improvement was 4.32%. In addition to this, it is clearly visible from both results that the selected 11 samples reflected the whole set of 41 samples reasonably well (76.83% – 78.1%).

Conclusions

We have developed a data-parallel GPGPU based region growing algorithm with improved seed point search and region growing iteration that is equally as accurate as the original sequential version, but it is suitable to process more than one cell nucleus at the same time, therefore its speed is 25–65% faster than the original CPU implementation. We have implemented the algorithm in CUDA environment (using Nvidia Fermi based GPGPUs). A possible improvement could be to implement the parallelization on a higher level, and develop the system so that it can be executed simultaneously on multiple GPUs (and in addition, on multiple CPU cores).

In the next step we have developed an evolution-based algorithm to find the optimal parameters for this new data-parallel version of the region growing method. This genetic algorithm was used to successfully determine a set of parameters that could be used to achieve 81.15%, which means an improvement of 4.32% compared to the previously known best set of parameters.

One direction for further developments could be towards the specialized search techniques, because it is possible to arbitrarily select which tissue samples the clients should use for the evaluation and verification; so it might be feasible to narrow down this selection to include only the tissue samples of a given type, and this way it might be possible to find an even better set of parameters for the developed algorithms.

We have developed a new method (based on pixel level and object level comparisons) which is suitable to evaluate the accuracy of cell nuclei detector algorithms. In the next step we have successfully implemented this using an improved backtrack search algorithm to process high number of overlapping cells, which requires significantly fewer steps than the traditional linear search methods.

We have already developed a distributed framework for the execution of the genetic algorithm. The framework have lived up to our expectations, the execution time of 440 generations was fully acceptable.

References

[1] Z. Shebab, H. Keshk, M. E. Shourbagy, “A Fast Algorithm for Segmentation of Microscopic Cell Images”, ICICT '06. ITI 4th International Conference on Information & Communications Technology, 2006, ISBN: 0780397703

[2] J. Hukkanen, A. Hategan, E. Sabo, I. Tabus, “Segmentation of Cell Nuclei From Histological Images by Ellipse Fitting”, 18th European Signal Processing Conference (EUSIPCO-2010), 23-27 Aug. 2010, Denmark, ISSN 207614651219, pp. 1219-1223.

[3] R. Pohle, K. D. Toennies, “Segmentation of medical images using adaptive region growing”, 2001, Proc. of SPIE (Medical Imaging 2001), San Diego, vol. 4322, pp. 1337-1346.

[4] Pannon University, “Algoritmus- és forráskódleírás a 3DHistech Kft. számára készített sejtmag-szegmentáló eljáráshoz”, 2009

[5] L. Ficsór, V. S. Varga, A. Tagscherer. Zs. Tulassay, B. Molnár., “Automated classification of inflammation in colon histological sections based on digital microscopy and advanced image analysis”, Cytometry, 2008, pp. 230–237.

[6] S. Szenasi, Z. Vamossy, M. Kozlovszky, “GPGPU-based data parallel region growing algorithm for cell nuclei detection“, 2011 IEEE 12th International Symposium on Computational Intelligence and Informatics (CINTI), 21-22 Nov. 2011, pp. 493-499.

[7] I. J. Rudas, J. K. Tar, “Computational intelligence for problem solving in engineering”, 36th Annual Conference on IEEE Industrial Electronics Society (IECON), 7-10 Nov. 2010, pp. 1317-1322.

[8] S. Szenasi, Z. Vamossy, M. Kozlovszky, “Evaluation and comparison of cell nuclei detection algorithms“, IEEE 16th International Conference on Intelligent Engineering Systems (INES), 13-15 June 2012, pp. 469-475.

[9] S. Sergyán, L. Csink, “Automatic Parameterization of Region Finding Algorithms in Gray Images”, 4th International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 17-18 May. 2007, pp. 199-202.

[10] Goldberg, David G, “Genetic Algorithms in Search, Optimization, and Machine Learning”, Addison-Wesley Longman Publishing Co., Boston, ISBN: 0201157675

[11] C. Carrick, K. MacLeod, “An evaluation of genetic algorithm solutions in optimization and machine learning”, 21st Annual Conference Canadian Association for Information Science CAIS/ACSI’93, Antigonish, 12-14 July 1993, pp. 224–231.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, “Introduction to Algorithms”, Mcgraw-Hill College, 1990, ISBN: 0070131430, Chapter 17 "Greedy Algorithms", p. 329.

[13] Z. Blázsik, Cs. Imreh, Z. Kovács, “Heuristic algorithms for a complex parallel machine scheduling problem”, Central European Journal of Operations Research, 2008, pp. 379-390

[14] J. L. Noyes, “Artificial Intelligence With Common Lisp”, D.C. Health and Company, 1992, ISBN: 0669194735, pp. 157-199.

[15] S. N. Sivanandam, S. N. Deepa, “Introduction to Genetic Algorithms”, Springer, 2008, ISBN: 9783540731894

[16] M. Gendreau, J.-Y. Potvin, “Handbook of Metaheuristics”, Springer, 2010, ISBN: 1441916636

[17] D. Bersimas, J. Tsitsiklis, “Simulated Annealing”, Statistical Science, vol. 8, no. 1, Institute of Mathematical Statistics, 1993, pp. 10-15.

[18] D. Whitley, “A genetic algorithm tutorial”, Statistics and Computing, vol. 4(2). 1994, pp. 65–85.

[19] C. Reews, “Genetic algorithms”, Coventry University, Heuristics in Optimization course, 2012

[20] M. Mitchell, “Genetic algorithms: An Overview”, Complexity, 1995, pp. 31-39.

[21] Y.-S. Choi, B.-R. Moon, “Parameter Optimization by a Genetic Algorithm for a Pitch Tracking System”, Proceedings of the 2003 international

conference on Genetic and evolutionary computation: PartII, GECCO'03, 2003, ISBN: 3540406034, pp. 2010-2021.

[22] A. Bevilacqua, R. Campanini, N. Lanconelli, "A Distributed Genetic Algorithm for Parameters Optimization to Detect Microcalcifications in Digital Mammograms", EvoWorkshop 2001, 2001, pp. 278-287.

[23] L. Budin, M. Golub, A. Budin, "Traditional Techniques of Genetic Algorithms Applied to Floating-Point Chromosome Representations", Proceedings of the 41st Annual Conference KoREMA, Opatija, 1996, pp. 93-96.

[24] S. Szenasi, Z. Vamossy, M. Kozlovszky, "Preparing initial population of genetic algorithm for region growing parameter optimization", 4th IEEE International Symposium on Logistics and Industrial Informatics (LINDI), 2012, 5-7 Sept. 2012, pp. 47-54.

[25] R. Kohavi, F. Provost, "Glossary of Terms", Machine Learning vol. 30 issue 2, Springer Netherlands, 1998, pp. 271-274., ISSN: 08856125

[26] E. Tóth-Laufer, M. Takács, I.J. Rudas, "Conjunction and Disjunction Operators in Neuro-Fuzzy Risk Calculation Model Simplification", 13th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 20-22 Nov. 2012, ISBN: 9781467352048, pp. 195-200.

[27] Zs. Cs. Johanyák, O. Papp, "A Hybrid Algorithm for Parameter Tuning in Fuzzy Model Identification", Acta Polytechnica Hungarica, vol. 9, no. 6, 2012, pp. 153-165.

[28] M. T. Goodrich, R. Tamassia, "Algorithm Design: Foundations, Analysis, and Internet Examples", John Wiley & Sons Inc., 2002, ISBN: 0471383651

[29] M. Mitchell, "An introduction to genetic algorithms", Bradfork Book The MIT Press, Cambridge, 1999, ISBN: 0262133164

[30] D. Gupta, S. Ghafir, "An Overview of methods maintaining Diversity in Genetic Algorithms", International Journal of Emerging Technology and Advanced Engineering, vol 2., issue 5., May 2012, ISSN: 22502459, pp. 56-60.

[31] K. Messa, M. Lybanon, "Curve Fitting Using Genetic Algorithms", Naval Oceanographic and Atmospheric Research Lab., Stennis Space Center MS., NTIS issue number 9212.