Design and implementation of a parallel heuristic method for spatial aggregation problems

**Abstract**: Spatial aggregation refers to the grouping small spatial units such as census blocks into customized large ones. By using this, users can design their own suitable regions for different purpose without necessarily only using census data with imposed and fixed census boundaries. Spatial aggregation problems can be solved by many algorithms or methods. Among all these methods, a factor that is needed to consider is method’s performance and computational time. One of these methods is the Give and Take algorithm (GAT) that is effective and is relatively efficient in computation time. In general, GAT first randomly aggregates spatial units into a specified number of districts and then keeps randomly swapping spatial units between adjacent districts until a certain objective, such as population equality, is met. However, GAT has two limitations. First, GAT cannot always generate good solutions even using the same set of parameters. For some solutions, their objective function values cannot be further improved by increasing the total number of iterations when the total number of iteration reaches a certain point. Second, GAT is designed to generate a single solution instead of **multiple** solutions at one time. This paper presents an improved method to address these two limitations. The first limitation is solved by recombining two solutions into a new one and then improve that new solution. This recombining method simulates the process of an evolutionary algorithm. The second limitation is solved using parallel computing techniques. The approach and experiment are implemented and analyzed in this paper. The results from a series of computational experiments show that this new approach can be used to improve performance and save computation time.

**Key words:** Spatial Aggregation, Evolutionary Algorithm, GAT, Parallel Computing, Census

**Introduction**

Spatial aggregation, also called zone design, refers to the procedure of grouping n small spatial units into k (k<n) large zones such that some objectives are met and some spatial constraints are maintained. The objectives we want to meet from this type of problem may include total population equality, population rate equality and income maximization. Spatial constraints request spatial units to be internally homogeneous and to occupy contiguous regions in space [3]. One of the common application is to solve electoral districting problem, which aims to aggregate spatial administrative units like census tracts, into a predefined number of districts or regions such that each region is contiguous and the total population of each region is as similar as possible [4]. In this problem, the quality of population equality between each district is considered as objective function and **maintaining** contiguity of each district is considered as spatial constraint. This aggregated result has two advantages: first, researchers can use their own customized districts instead of only using the fixed boundaries that census bureau provides. Second, since each new districts maintains population equality, many researches could be done without worrying about possible side effect that population itself has brought about.

Spatial aggregation problems can be considered as geographic optimization problem. In general, geographic optimization problem can be categorized into four types [5]. The first one is the selection problem without spatial constraints. This type of problem finds the subset of spatial entities to satisfy one or more goals. One typical example of this problem is the p-median problem which is to locate p facilities to serve *n* demand points while minimize the total travel cost between facilities and their nearest demand points [6]. Densham, Rushton had done some research on this area and purposed a new efficient genetic algorithm to solve p-median problem [7].

The second type is the selection problem with spatial constraints. In addition to the selection procedures that are similar with the first problem type, it needs to maintain a set of spatial constraints. One common example of this is site selection problem which is often used in biodiversity protection. It not only needs to pick some protection sites but also enforces site contiguity. Other constraints, on the other hand, may need sites not be connected with each other. Much research has been done to solve this type of problem. In 2000, [Cova](https://scholar.google.com/citations?user=bnnYw7AAAAAJ&hl=en&oi=sra), [Church](https://scholar.google.com/citations?user=t4-JG2IAAAAJ&hl=en&oi=sra) formulated multiple spatial constraints for a single region site search problem [8] while in 2002, Önal, Briers incorporated spatial constraints and used integer programming method to solve this problem [9]. In 2005, Shirabe purposed a new formulation of contiguity that can be used to describe spatial constraints and be **incorporated** into integer programming models [10]. In 2009, Shirabe proposed a new integer programming approach to solve site selection problem [11].

The third type of problem is the partition problem without spatial constraints. Each spatial object is assigned a value and we need to find out the combination value of those spatial entities to satisfy some objectives. **Bennett**, Xiao, and Armstrong used Evolutionary Algorithm to produce optimal or near optimal solutions to solve land management problem [12].

The last type of this problem, which we will address specifically in this paper, is the partition problem with spatial constraints. In addition to the process of third type of problem, spatial constraints must be satisfied. One common spatial constraint requests spatial units to be internally homogenous and occupy contiguity regions in space. Political aggregation problem is exactly such a problem in which **spatial** units need to be aggregated into new regions while internal spatial units of each new region need to be maintained contiguous with each other.

Solving political aggregation problem is a challenge for two reasons. First, it is hard to formulate a general or standard criteria to guarantee spatial constrains like contiguity. Second, solving redistricting problem is computational expensive because the number of feasible solutions will increase exponentially with the input size [5], so it is impossible to exhaustively try all possible solutions and discard those which is not optimal. In this case, exact method is not suitable for solving this problem because it will systematically examine all possible solutions before it achieves global optimality. Instead, heuristic method are used to yield a good, but not necessarily optimal result to **achieve** near optimal solutions.

Many methods have been developed to solve political aggregation problem so far. In 1991, Openshaw and Rao tried to solve aggregation problem by using automatic zoning procedure (AZP) algorithm [13] and applied this method in the field of social economic units. AZP first generates a random solution and then randomly picks a region to let units move from one region to its neighbour without violating spatial constraints. It keeps going like this until no more improvement can be made. This method, though achieves good result, is computational expensive. Improvement to AZP lead to tabu search heuristic in 1995 which has optimized the spatial unit’s searching process to save some computational time [13]. In 2004, Bacao, Lobo, Painho applied generic algorithm (also called Evolutionary Algorithm) in zone design problem. Basically, they encoded each solution as a string bit by taking either the central of spatial unit or any place within that unit as one bit. Then by combining bits of two different solutions or randomly changing some bits, it generates new solutions. Finally, it uses spatial compactness and population equality as criteria to select good solutions and eliminate bad ones. In 2005, Assunção, Neves used Minimum Spanning Tree (MST) to regionalize social economic units. They transformed this aggregated problem into a graph partitioning problem and then used their own heuristic method to optimize the process of manipulating subdivide trees and searching optimize solutions which saves time. In 2008, Xiao proposed a unified framework to solve geographic optimization problem based on Evolutionary algorithm and graph theory. He described in details how graph can be used to represent spatial units and how specific evolutionary operations should be designed based on graph. In 2011, Kim utilized Xiao’s method and developed a new and efficient algorithm called Give and Take (GAT).Basically, it uses graph to represent spatial units and the connectivity relationship between each other. This algorithm first generates an initial solution by keeping randomly assigning a spatial unit to its nearby districts until no more units left. Then it begins to randomly swap spatial units between neighbouring districts without violating contiguity until an optimal objective like population equality is achieved or the number of iterations have reached to the predefined threshold. This algorithm performs effectively and efficiently in terms of its ability to aggregate spatial units while maintains population equality.

However, GAT does have some limitations. First, its final result is not always good even for the same parameter setting. For those results that are not that good, they cannot be improved by purely increasing the number of iteration. Second, this algorithm is designed to generate one good result at a time. If user wants to have more than one result, he/she should run this algorithm multiple times. This is not a big issue when dealing small dataset, but the factors of saving computational time becomes more critical when the input dataset becomes larger. This paper solves these two limitations by utilizing the thinking of evolutionary algorithm and the parallel computing technology. **Section** 2 describes the methods, showing how to use these method to improve performance and save computational time. In section 3, we apply methods on a set of experiments and make evaluation on their results and performance. Section 4 presents the conclusion and possible future work that needs to be done.

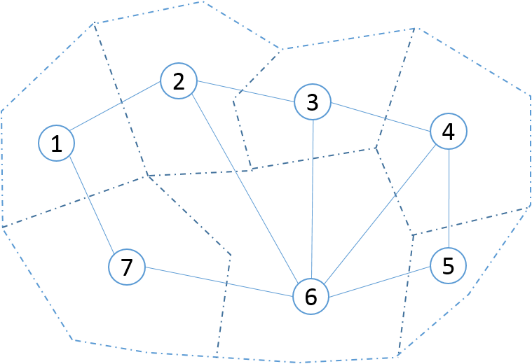
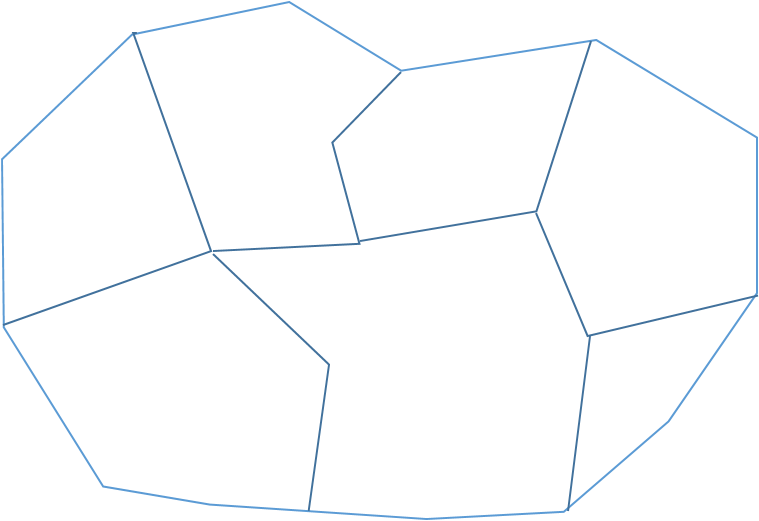
**Methods**

**This paper extends the GAT (Give and Take) algorithm, one of the heuristic methods that can be used to solve spatial aggregation problems using a recombination method and parallel computing techniques. These three methods will be discussed and explained in details in this section.**

**Graph Theory and its representation**

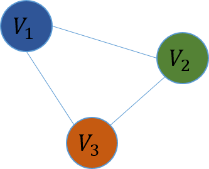
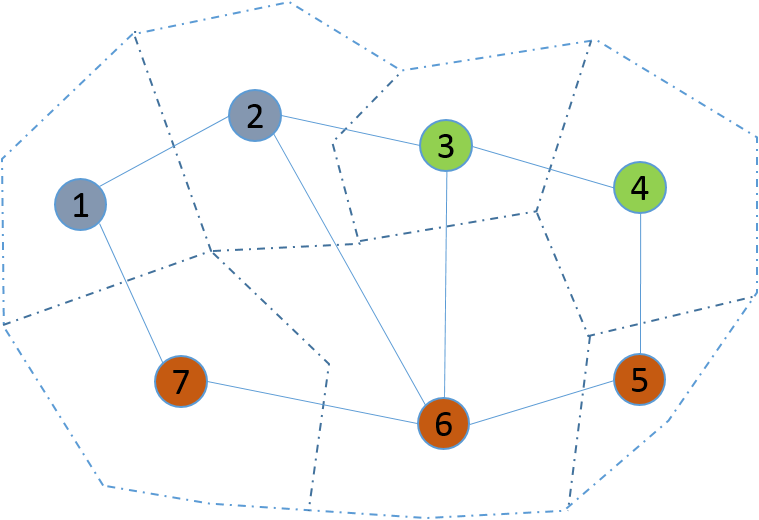
**A graph is defined as where is a set of verities or nodes and is a set of edges that may connect one vertex to another. For example, can represent an edge between vertex and vertex. A graph can be categorized as directed and undirected graph. Directed graph has a direction attribute for each edge while an undirected graph does not have such attribute.**

**In this paper, graph is used to represent spatial units. Each census tract or block in a map is represented as a vertex or node in a graph. Edge between each vertex in a graph is used to represent the contiguity between each spatial unit. Since no direction attribute is used, I will use undirected graph. The relationship between geographic map and its corresponding graph representation is shown in figure 1.**



***Figure 1:The left part is geographic map in real world and the right part is its corresponding graph representation. Here V={1,2,3,4,5,6,7} and E= {(1,2),(1,7),(2,3),(2,6),(3,4),(3.6),(4,5),(4,6)}.***

**Now all the process will be based on manipulating graph. It is obvious to see that the result or the solution will also be a graph. The only difference of the result graph compared with original graph is that for each node or vertex, it has a label to specify which group or new region that node belongs to. One example of the result using graph representation is shown in figure 2. Every node in the new graph has a unique colour to label its group. This new graph can be further abstracted. For each new group, it can be considered as a new ‘super node’ which contains more than one ‘sub node’ in original graph and I can use ‘super edge’ to represent the contiguity between each group. Then in this case, I can create a new super graph which simplifies the graph structure while maintaining core aggregation results and spatial relationship between aggregated groups. The simplified super graph will be used in solution recombination process which will be discussed later.**



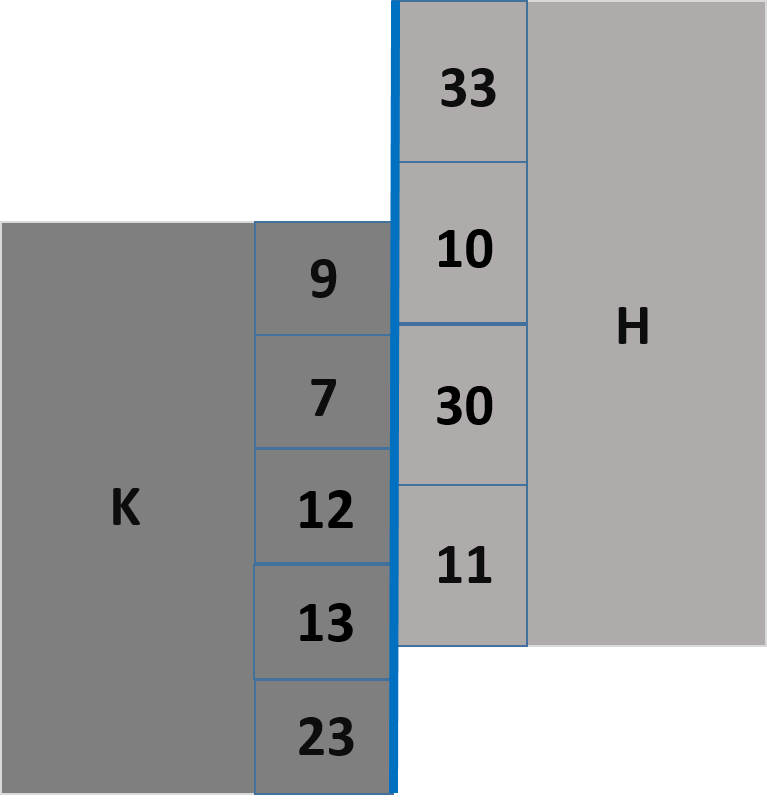
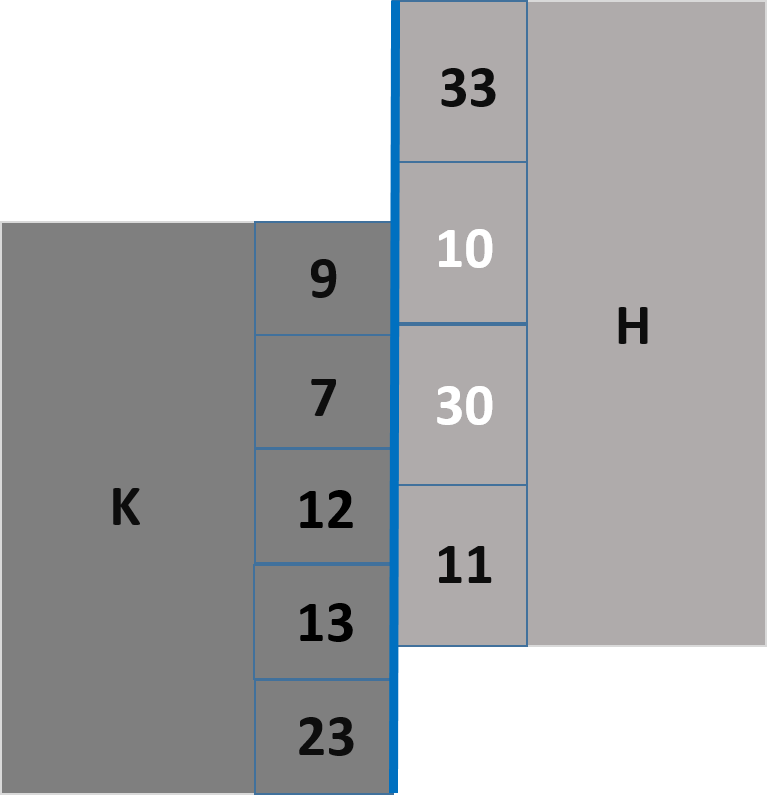
***Figure 2: The left part is one partition result example. The graph is divided into three groups. The right part figure is a simplified super graph which only maintains the grouping result and the contiguity relationship between each group. Here, V= {}, E = {(), (), ()} while.***

**GAT Algorithm**

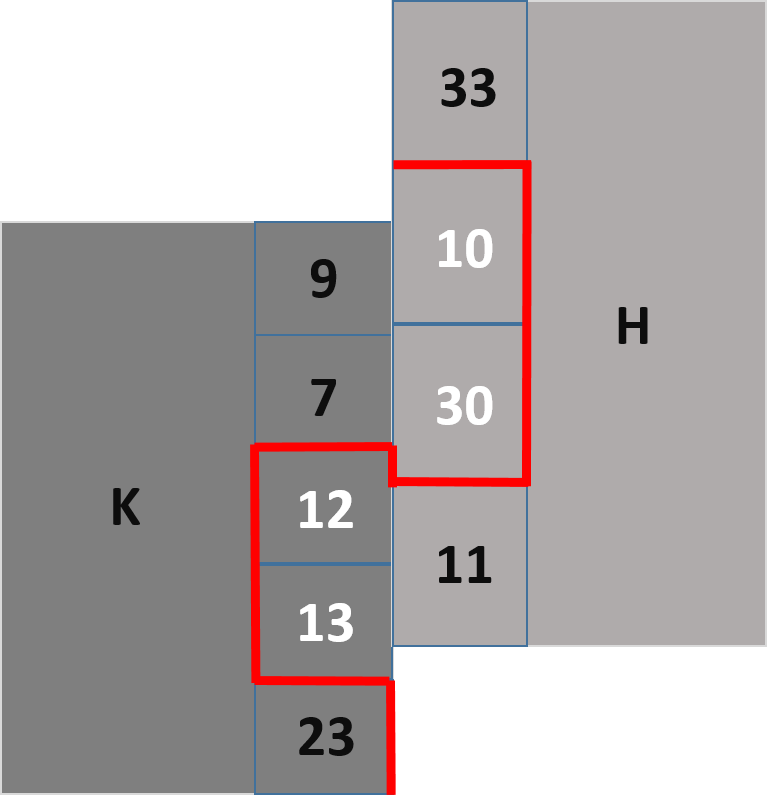
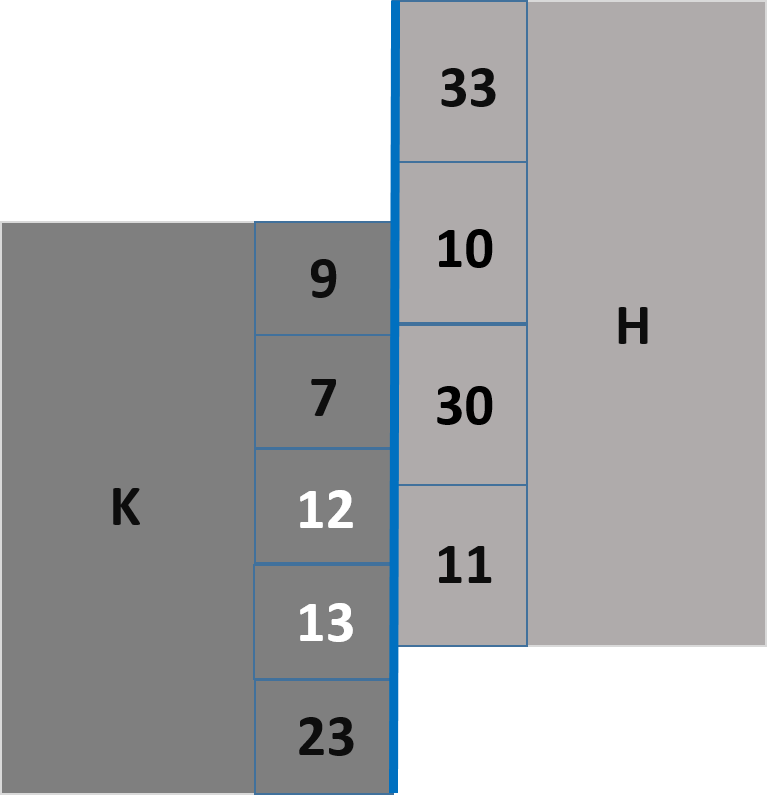
The GAT algorithm is first developed by Kim [2] in her PhD dissertation. It is a new heuristic method as well as greedy algorithm that can be used effectively in the political redistricting problem. In her dissertation, Kim used population equality as the objective and spatial contiguity as a spatial constraint. The algorithm uses graph to represent spatial units and all the optimized process is based on manipulating graph. I will first introduce GAT’s general idea and then dig into details of this algorithm.

***General idea:***

Suppose we have an original district as figure 3 (I) shows. District K and District H have different population with a gap of 20 and we want to at least minimize this gap (if not possible to set this gap to zero) between the two districts. We first pick a group of census units from high regions (here we choose units with population 10 and 30 from *H*) and move these units into low regions (here is *K, see figure 3 (II)*). The total population of these moving units should be larger than the population difference such that after moving, the population of region *H* now is less than the population of region *K*. Then we repeated taking single census unit back from *K* to *H* to make complement of that population gap until the total population that returned (the difference between population added and population returned) is less than initial population difference (here we choose units with population 10 and 30 from H and return these units back, see figure 3 (III)). After this swapping, we find out that the population gap between district *K* and *H* has decreased (see figure 3 (IV)). If this is the case, we call this swapping process a *suitable swapping* (this will be used later) because this swapping has made the population gap smaller. In GAT, it iterates this process until no more improvement can be made or it reaches the maximum number of iterations that user sets for it.

***(I) (II)***



***(III) (IV)***

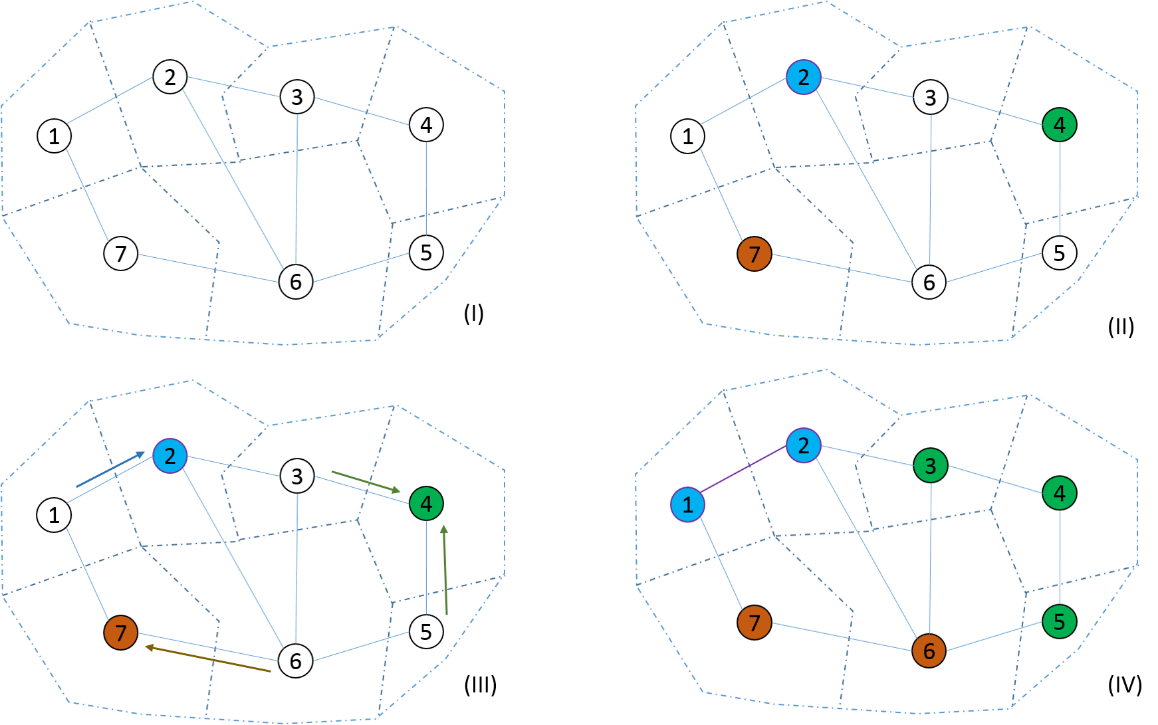
***Figure 3:*** *(I): Original census map with region K and region H. Total population in K is less than total population in H. Differences= Pop(H)- pop(K) = 20. (II): Census units with pop 10 and 30 from region H is selected and move from H to K. The total giving is 40. (III): Census units with pop 12 and pop 13 is selected from region K and return from K to H. The total return is 25. (IV): Result with regions and now the gap between K and H is 5 which is decreased.*

***Implementation:***

The GAT algorithm consists of two major two steps: Initialization and Optimization.

* **Initialization**

The first step is to roughly generate an initial solution. Suppose we want to aggregate the original census units into *k* parts. First we randomly pick k unit as seeds for each new group. Then we keep randomly assigning spatial units all the units are assigned. Since the whole process is random, it is hard to guarantee population equality in initialization. The initial solution is used to be further improved in optimization process. The workflow of figure 4 shows this initialization process.



***Figure 4: (I). Original graph with 7 spatial units at the beginning. Suppose we want to aggregate this graph into three new groups. (II).Randomly pick three nodes as basic for new groups. (III). for each other nodes, randomly assign them to their nearby group. In this example, node 1 is assigned to blue, node 6 is assigned to group brown and node 3 and 4 are assigned to group green. (IV). Initial solution is generated.***

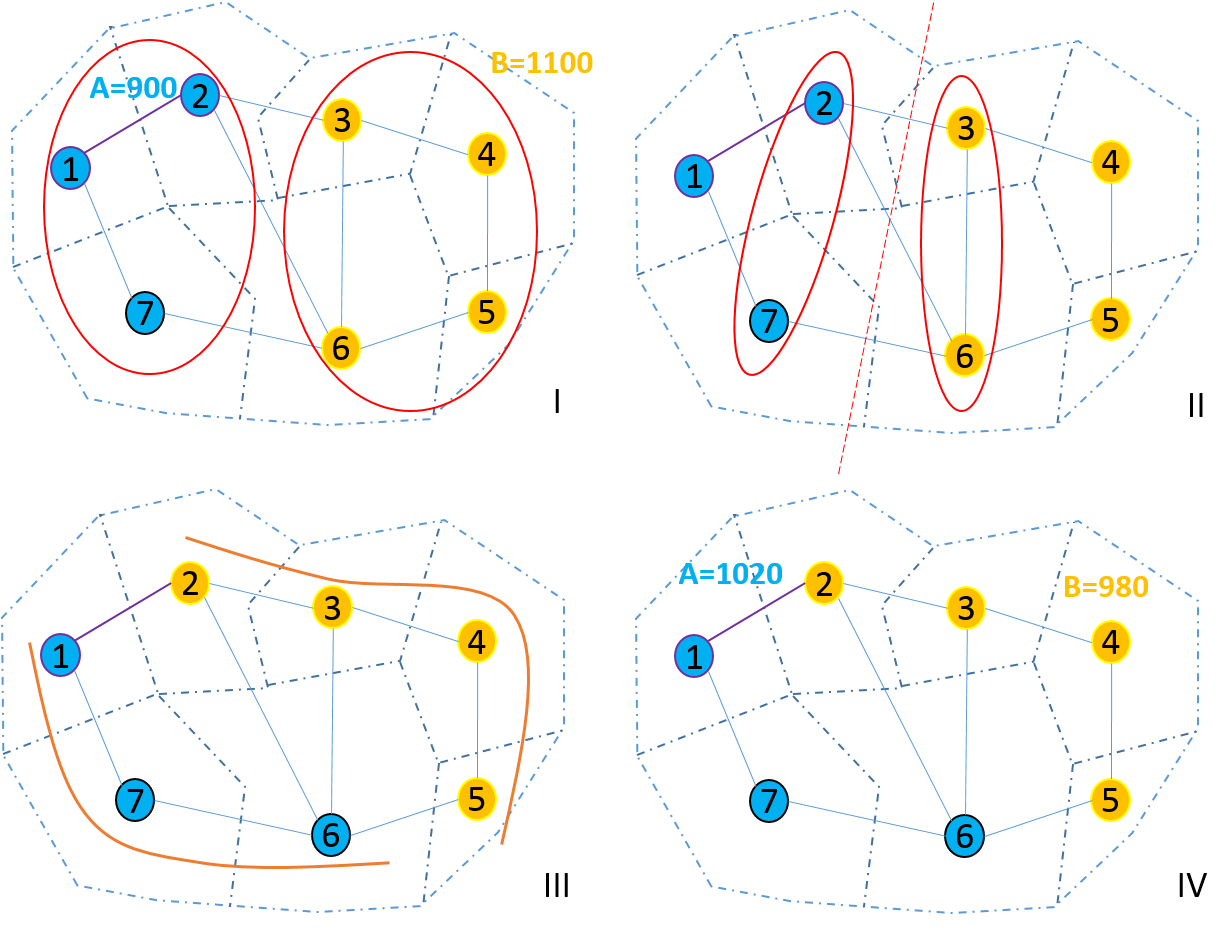
* **Optimization**

After having an initial solution, GAT is applied to optimize this result by swapping census units that belong to groups with different population but are adjacent with each other. To be more specific, first, it randomly picks two groups whose population are not equal but are adjacent with each other. I call the group with lower population LOW\_D and group with higher population HIGH\_D for convenience. We calculate their population difference (called diff1). Second, we find the border between these two groups. The border here contains units that can be swapped from one group to another. Among the units within the border, the algorithm finds all units that can be swapped from HIGH\_D to LOW\_D (I call these units a set uH) and all units that can be swapped from LOW\_D to HIGH\_D (I call these units a set uL). Now we know that uH and uL are the sets that contains spatial units that need to be swapped. Third, we do the following steps many times until we find a *suitable swapping:* in step one, we randomly pick random number of units from uH that are going to move from HIGH\_D to LOW\_D (I call it uH\_selected) and calculate the population to be added. In step 2, we iteratively picking one unit from uL and give it back to HIGH\_D until returned population (added population – population given back) is less than differ1.

After a *suitable* *swapping* is done, we will check whether new graph still maintains spatial contiguity after swapping. If not, this swapping is not valid, we will redo the process until it finds the *suitable swapping*. According to Kim, the method of checking validation of spatial contiguity follows three steps:

1. Randomly select a unit among those swapping units as a seed.
2. Based on this seed, keep adding adjacent units into this seed lists until no units can be added.
3. Calculate the total number of units that has been included into that seed lists (including the seed itself). If its number equals the total number units in that district, then this seed can be swapped.

The workflow of figure 5 shows this process and figure 6 is the corresponding pseudo code.

***Figure 5: (I). Randomly pick two groups to be swapped. Group A with pop 900and group B with pop 1100 is selected. LOW\_D=A, HIGH\_D=B and diff1=200. (II). Find the border between A and B. Based border, node 2 and 7 can be swapped from A to B and node 3 and 6 can be swapped from B to A. So uL={2,7}, uH={3,6}. (III). suppose node 2 with pop 80 from uL is picked and node 6 with pop 200 is picked from uH and swapped. In this case, uL\_selected={2}, uH\_selected={6}. (III).Contiguity check and it is valid based on the rule. (IV). Final new solution. Group A with pop 1020 and group B with pop 980. Pop gap is 40 < 200.***

**Input:** A Graph which represent the real world census map; number of regions that need to be aggregated; number of iterations that needs to be run.

**Output:** Aggregated results and objective function value.

Initialization ()

For each iterated solution :( # of iterations):

For each district,put all the district whose pop< target into the list\_low:

Randomly pick one from the list\_low, called LOW\_D:

Put all the districts which adjacent to LOW\_D and whose pop>target into the lists\_high

If there is no such district, return

Else

Randomly pick one from the lists,called HIGH\_D

Calculate the High pop from HIGH\_D

Calculate the Low pop from LOW\_D

Calculate the diff( called diff1)

Find the edges on border(units can be swapped)

uH=Find the border nodes belong to HIGH\_D

uL=Find the border nodes belong to LOW\_D

Swap=FALSE

While (! swap OR uH is not empty) do:

uH\_selected=randomly pick many from the uH

Remove the picked from uH

Calculate the pop from uH\_selected(actuall pop to be added)

While (returned pop<differ1 OR uL is not empty) do:

u=randomly pick one from uL

Remove the picked one from uL

Calculate the gap(gap=returned pop-differ1)

If gap>0:

Add the u into the uL\_selected

If gap<0:

Swap=DONE

If swap==DONE:

Add nodes uL\_selected to uH

Add nodes uH\_selected to uL

Remove nodes uL\_selected to uL

Remove nodes uH\_selected to uH

Reset graph

Valid\_solution()

If validated:

return result

Else

return NONE

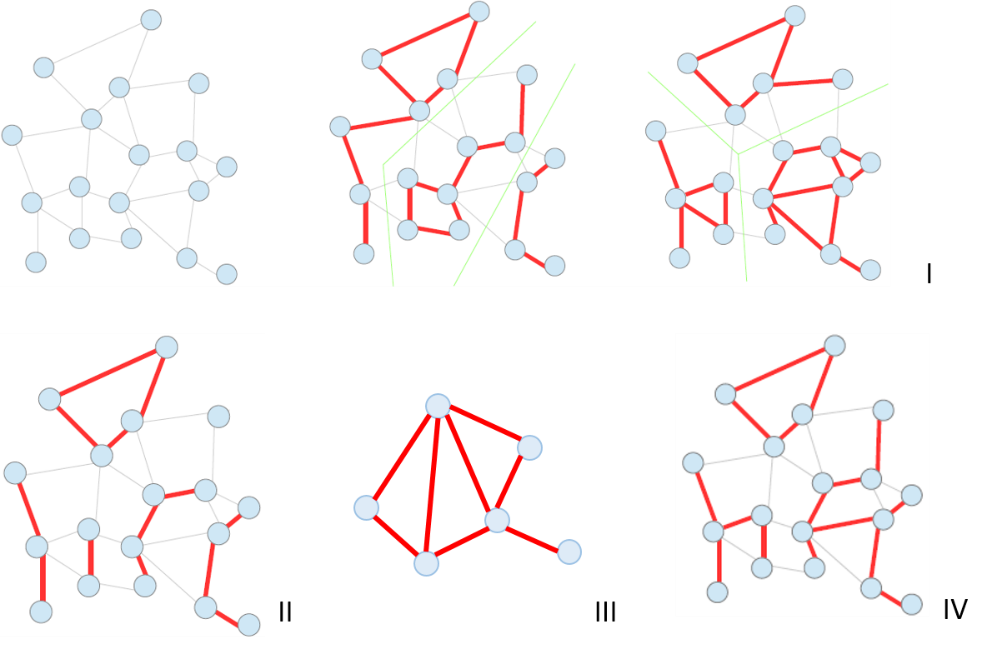
***Figure 6:pseudo-code of GAT algorithm***

**Solution Recombination**

Normally, with the number of total iteration increasing, the final result is getting close to our objective. But that only happens within a certain small range of iterations. When the total number of iteration reaches a certain point, the final result cannot be further improved no matter how many iterations will be added. Sometimes, due to the randomly selection process, the result may even get away from our objective by constantly increasing its number of iteration. That’s one of the limitations GAT has when applied to solve aggregation problems. Solution recombination method is designed to solve this limitation.

The main reason why the result cannot be further improved after its iteration reaching a certain point is that the overall structure of that aggregation has already reached its local optimal point and there are not much valid or suitable swapping left for that solution. In other words, any additional iterations won’t change much of its structure. One way to change this situation is to use some methods to incorporate new graph structure from other solutions in order to let suitable and valid swapping emerge again. Recombination does this job by combining two already existing solutions into a new one such that the new structure can be further improved by using GAT again. This process is similar with evolutionary process when gene can be evolved into even better next generation by incorporating with different organization of other gene.

Suppose we have two solutions for the same census map (see figure 6 (I)) and we want to combine these two solutions into one and apply GAT again to produce an even better solution which GAT itself cannot do. We first find the common edges of these two solutions and these edges are kept while the rest of them are discarded. After this, a new graph is created and its number of groups is greater than the number of groups we want to aggregate (see figure 3a (II)). Here each group can be considered as a super node. If there exists an edge between two groups in their original graph, then there exists a super edge of these two super node in their super graph. The population of that super node is the total population of nodes existing in that group. To construct a valid solution, we always merge the super node with smallest population to its neighbouring super node with the smallest population. It keeps going like this until the total number of super node is equal to the number of group we want to aggregate. Now this super graph can be converted back to its original graph which new useful information have been incorporated from both solutions. Finally, GAT is applied to this new graph again to produce an even better result. Figure 3a shows the whole process. Detailed experiment and analysis will be elaborated in section 3.



***Figure 6: (I). Initial graph and its two solutions. Each subgraph represents one group. I assume I want to have three new districts which is three groups. (II). the subgraph after finding common edges between solutions. Now it has more than three subgraphs. (III). the result of super graph. Consider each subgraph as super node and if there exists an edge between two groups in their original graph, there exists a super edge between two super nodes in new super graph. (IV). Result after merging. Then this graph with new structure can be used GAT again to generate better solutions.***

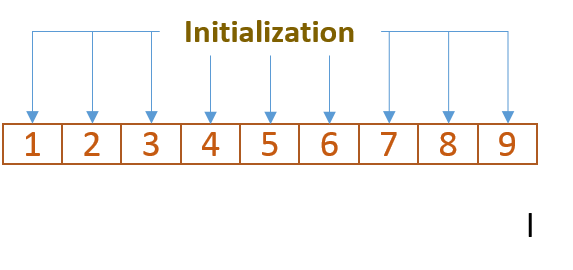
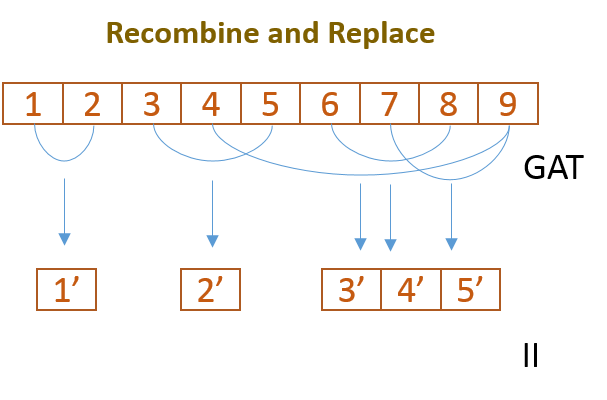
**Multi-solutions and Parallel Computing Technology**

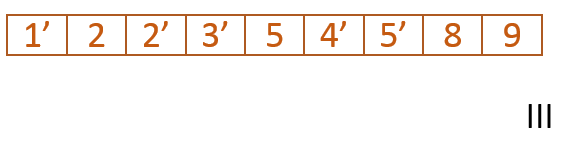
Another limitation of GAT is that it can only generate one solution at a time. In order to produce multiple good solutions, user has to run the program multiple times. Running the program multiple times may not take long when dealing with small datasets. But it becomes hard or even impossible to run the program multiple times to process large datasets because even a single time may take a long time. Parallel computing techniques here is used to solve this problem. By using this technique, it can produce multiple solutions at the same time which greatly reduce the computational time.

Some prerequisites of the parallel algorithms need to be prepared before digging into details of this idea. First, a pool is used to hold multiple solutions (can either be initial solutions or optimal solutions). GAT algorithm, solution recombination and parallel computing will then be used to process the data that stored in pool. Second, a computer with multicore is essential to let parallel computing techniques work.

There are a few advantages to use a pool to store multiple solutions and to use a parallel program to access the pool directly instead of dealing with a single solution like traditional GAT. First, since the pool can store many solutions at the same time, when applying GAT algorithm combined with parallel computing to optimize it, a user can get a pool of good results instead of just one good result. Besides, since we use parallel technique, many solutions can be generated and optimized in a parallel way such that the total amount of time it takes to generate many results is almost equal to the previously time that generates one result. Of course, time will sometimes be longer due to parallel computing communication and the limitation on number of cores a computer has. Thirdly, since we have a pool of solutions, information on different solutions is abundant. When applying to solution recombination, there are a lot of new candidate solutions that can be selected to recombine, thus getting more chance of generating even better result. Of course, like gene evolution, combination also takes risk of generating bad solutions because we cannot control which information to incorporate and some information from other solutions may make the result get worse. In this case, we still simulate the process of evolution: discard them and generate the recombining result again. The later sensitive experiment shows that 60% of the recombining result is better than the original result at first time they recombine, thus 60% of them don’t need to be discarded and do the recombination again. Overall, this parallel algorithm save time while producing a pool of optimized result.

The detailed workflow for combining all these three methods is described as follows: First, I generate a pool of initial random solutions using parallel computing technique (figure 7 (I)). Then solutions in this pool are randomly chosen to be recombined and then GAT is used to improve the recombined solutions (figure 7 (II)). Recombined and GAT process can also be run in parallel way. Finally, if the new result is better than the either of original one, replace original result with the new result. (Figure 7 (III)) The figure 7 demonstrates the whole process.



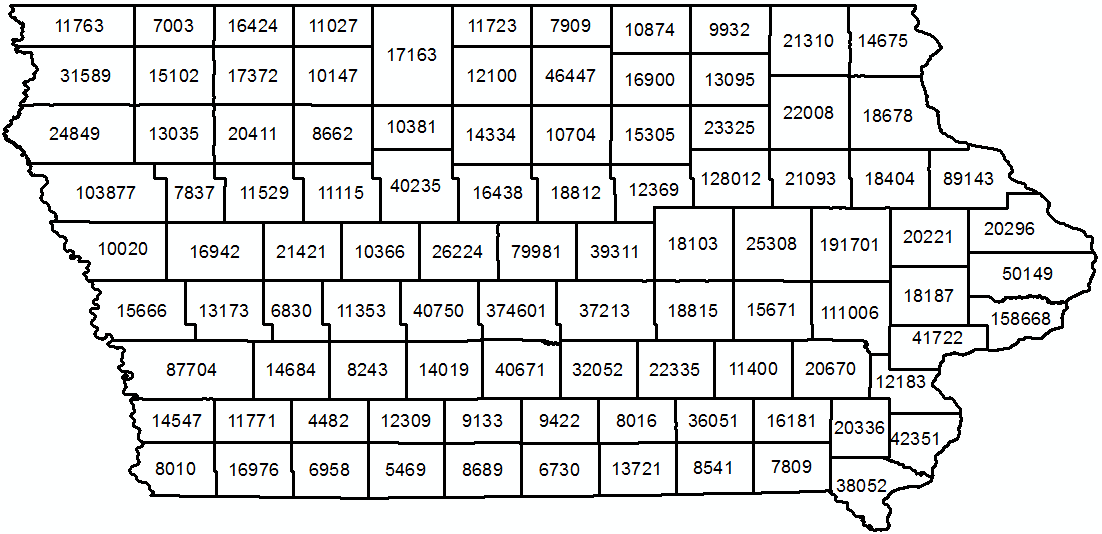
***Figure 7: Overall process from beginning to end. (I). Initialization process to generate 9 solutions labelled from 1 to 9. 9 solutions are generated at one time. (II).For each parallel thread, it randomly pick two solutions and do the recombining process. GAT is then applied to that recombined one to generate new solution. If new solution is better than the worse solution between two original solutions, replace the worse solution with new one. Here, new better solutions are 1’, 2’, 3’, 4’, and 5’. (III). Final replace result.***

**Experiment and Analysis**

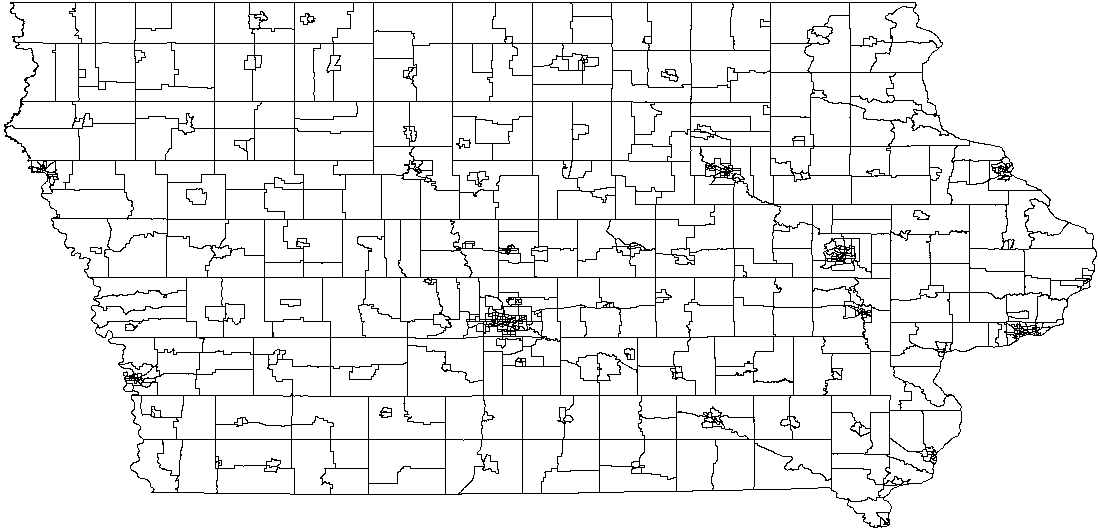
**Prerequisite**

* **Dataset**

The region that chosen to conduct experiment in this paper is Iowa State, USA. Two levels of the data are used: Iowa census tract data and Iowa county level data (see figure 8 and 9).These data can be downloaded from United State Census Bureau’s website (<http://www.census.gov/geo/maps-data/data/tiger.html>). Besides, US demographic data (<http://factfinder.census.gov/faces/nav/jsf/pages/index.xhtml>) which also comes from United State Census Bureaus is used to acquire population information.



***Figure 8: Census county data, 99 regions with pop labelled on each region.***



***Figure 9: Census tract data, with total of 825 sub regions. Due to too many regions, pop is not labelled on the map for better visualization.***

* **Programming environment**

The whole program is written in python version 2.7.2. The type of CPU is Intel(R) Core(TM) i7-4700MQ and it has 4 cores with 8 logical processors. It is a 64-bit Operation system.

* **Objective function**

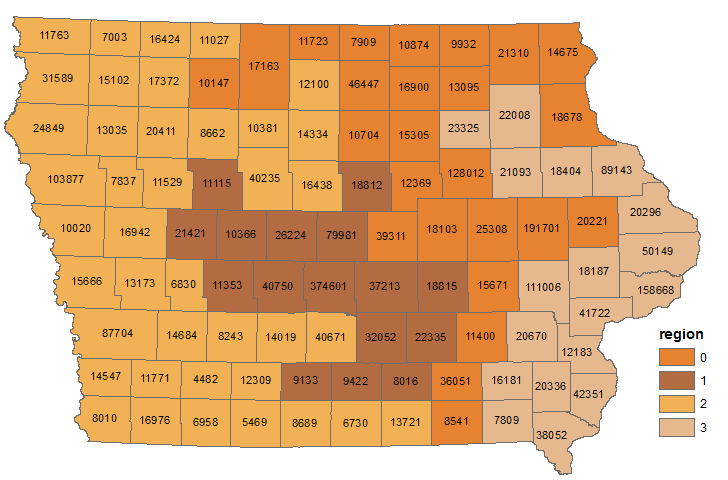
Objective function is used to evaluate the effectiveness of this algorithm. The objective function used in this paper is as follows:

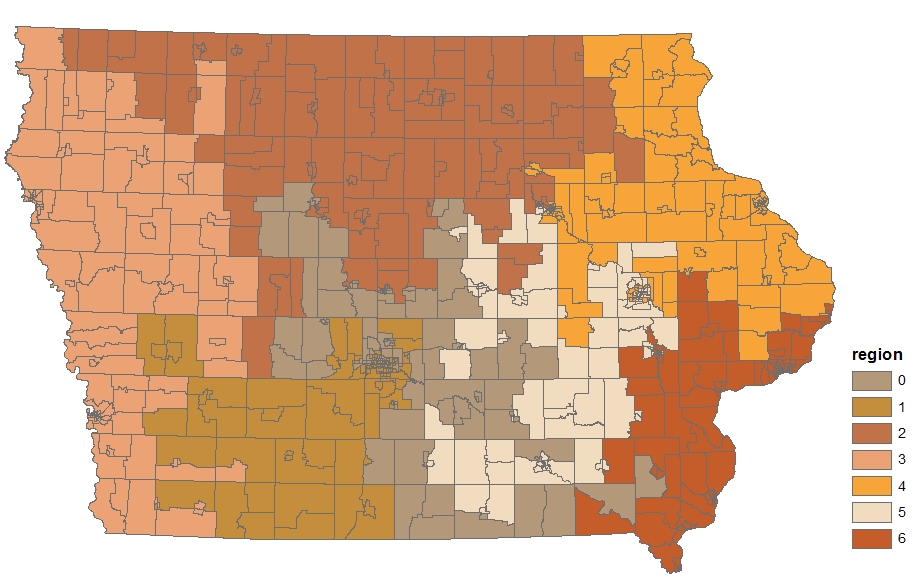
Where *P* is the total population, *n* is the number of district that user want to generate, is census unit’s population and is the ideal population which can be calculated by *P* divided by *n*. Ideally, this value should be zero which is hard to get. Our optimized goal is to make the value of objective function as small as possible.

**Aggregation Result**

The aggregation result depends on the following parameters: number of new regions user wants to aggregate, total number of iterations, size of the pool, the number of recombining times and the number of units that need to be aggregated. Objective function value and computational time also varies depends on these parameters. In this paper, I first fix these parameters to make analysis of experiment itself in the part of aggregation result, recombining solution and parallel computing evaluation. Systematically analysis of how different parameters will have effect on the final performance will be done after that. Finally, sensitively evaluation of this method will be done.

Figure 10 shows the result from both Iowa census tract and Iowa county data. Figure 11 is the corresponding population table.





***Figure 10: Upper part, aggregate census county data into 4 parts with 1000 iteration. ; Lower part, aggregate census tract data into 7 parts with 1000 iterations.***

|  |  |  |
| --- | --- | --- |
| Region | Count\_region | Population |
| 0 | 25 | 731550 |
| 1 | 16 | 731609 |
| 2 | 40 | 731582 |
| 3 | 18 | 731583 |

|  |  |  |
| --- | --- | --- |
| Region | Count\_region | Population |
| 0 | 109 | 433128 |
| 1 | 117 | 433183 |
| 2 | 133 | 433182 |
| 3 | 129 | 433186 |
| 4 | 119 | 433183 |
| 5 | 106 | 433181 |
| 6 | 112 | 433171 |

***Figure 11: left table, Census County population aggregation result with objective function value 0.0021. Right part, census tract population aggregation result with objective function value 0.00103.***

The result shows that, for each region, GAT tries its best to equalize population equality while maintains spatial contiguity. In practice, spatial contiguity between two units means there are at least two p`oints overlapping with each other in these two units’ common boundary. Since the swapping process is random, one of the negative effect of GAT is that it cannot guarantee a more reasonable new region shape despite its good performance. But multi-solution generation in this paper mitigates such a negative effect because there are many alternatives for user to choose from pool.

As mentioned previously in this paper, GAT’s objective function value of bad result cannot be further improved when the total number of iteration has reached to a certain point. Figure 12 shows the trend between number of iteration and objective function value.

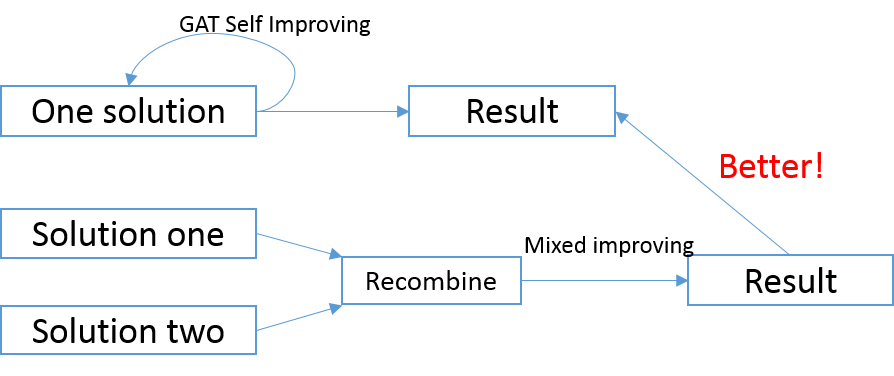
***Figure 12: before the number of iteration reaches to 1000, GAT works fine. However, after that, the improvement becomes slow. After 1100 iteration, the objective function value remains the same and cannot be further improved.***

**Solution Recombination Experiment**

Since only by increasing the number of iteration cannot further improve the final result when that number of iteration has reached to a certain point, more mixed information or graph structure needs to be used to gain more chance of generating new and better results. Solution recombination is designed for this purpose. As mentioned early, the experiment designed here will fix parameters including number of iterations, number of regions to be aggregated, pool size. Based on this prerequisite, the experiment is designed as follow: First, it generates two copies of initial solutions from the pool. Second, for these two solutions, one is used to run pure GAT while the other is run both recombination and GAT. Finally, I compare these two final result to see the difference. In order to compare them at the same level, the whole comparison process guarantees that the total number of iterations that pure GAT has is equal to the total number of iterations that recombination and GAT have. (See figure 13 for workflow.) Performance of both individual solution and a pool of solutions are tested and compared.

* **Single Solution Evaluation**

10 groups of single solution are tested on Iowa county data with 800 iteration and 4 aggregated regions fixed. For each group, there are two initial solutions. Pure GAT is used to give additional iterations to see whether it can further improve the performance. Recombined method is used to combine two initial solutions and use GAT to improve that combined solution. Figure 14 shows the result.



***Figure 13: Experiment workflow***

|  |  |  |  |
| --- | --- | --- | --- |
| Sol One | Sol Two | Add Iteration | Combined Add |
| 0.044151 | 5.765185 | 0.447456 | 0.007245 |
| 0.029320 | 0.134845 | 0.051054 | 0.035608 |
| 0.051122 | 0.145985 | 0.143115 | 0.068413 |
| 0.051669 | 0.068619 | 0.039093 | 0.036633 |
| 0.104978 | 0.261420 | 0.05618 | 0.008201 |
| 0.177561 | 3.287127 | 0.177561 | 0.125618 |
| 0.111061 | 5.294287 | 0.111061 | 0.102859 |
| 0.064791 | 12.618288 | 9.599757 | 0.031712 |
| 0.047021 | 2.523302 | 0.237363 | 0.019547 |

***Figure 14: single solution evaluation. For two initial solutions, Pure GAT is applied to add iterations for that worse solution while recombined method combine two solutions and apply GAT after that. By comparing two methods, we find that pure GAT cannot further improve the objective function value or cannot improve that much compared with combined solution method.***

* **Multi-Solutions Evaluation**

Now it is time to evaluate the overall pool performance instead of single group solution by using pure GAT and Recombined method. Since there are multiple solutions in one pool (which means there are multiple objective function value), I use mean, median and standard deviation value to demonstrate the overall performance of that pool. In this experiment, I set pool size to be 10, recombine times to be 50 and number of iteration to be 800. The initial pool, only iteration pool and recombining pool is show in figure 15. The mean, median and standard deviation value is shown in figure 16.

From the value comparison (see figure 17), we can see that although both pure GAT and recombined method have made the overall result of that pool better, recombined one has less mean and median and standard deviation value which means the overall pool performance is better when applied recombined method compared with only GAT method. That’s one of the improvement this paper has made.

|  |  |  |
| --- | --- | --- |
| Initial Pool | Pure GAT | Recombined |
| 0.509308 | 0.002119 | 0.003827 |
| 0.609707 | 0.003212 | 0.005468 |
| 1.131522 | 0.003759 | 0.005741 |
| 1.418298 | 0.008885 | 0.006219 |
| 5.70757 | 0.0095 | 0.006493 |
| 6.549104 | 0.017086 | 0.006766 |
| 13.27078 | 0.021529 | 0.0095 |
| 17.01548 | 0.023306 | 0.011892 |
| 22.87143 | 0.028637 | 0.013396 |
| 64.85276 | 0.492085 | 0.019752 |

***Figure 15: Pool value from Initialization, Only iteration and combined method.***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Mean | Median | Std |
| Initial | 13.3936 | 6.128337 | 18.655105 |
| Pure GAT | 0.061012 | 0.013293 | 0.143958 |
| Recombined | 0.008905 | 0.006629 | 0.004608 |

***Figure 16: Mean, Median and Standard deviation from Initial, Pure GAT, and recombined method.***

***Figure 17: Recombined Method’s mean, median and standard deviation value is less than Pure GAT method in terms of the overall performance in a pool of solutions.***

* **Sensitive Analysis**

GAT algorithm, due to its internal randomly swapping procedures, is not stable and cannot be guaranteed to generate optimal result every time. This side effect will also be brought into recombined method which makes recombining and applying GAT process cannot generate a better result every time. For the multi-solution generation, if the generated combined result is not better than either of those two selected solutions, the program will discard it and regenerate that again. In this case, this part may bring in some extra computation time. Sensitive analysis here is to determine how frequent this would happen. In this experiment, 20 times testing recombination process are run, and 11 of them generate better results at first time. This is not that good but it won’t have much effect on our final result of objective function value because the program will automatically discard that and do the recombine process again. As mentioned, the only side effect is that it will to some degree increase some extra computational time. Since we use parallel computing technology (which will be analysed later), overall, it still saves much time and make improved results compared with original pure GAT method.

**Parallel Computing Evaluation**

The second limitation of GAT is that it can only generate one solution at a time. Parallel computing is used to solve this problem while saves much computational time. The reason why it saves time is that multiple solutions will be generated from pool at the same time instead of through multiple iterations. This part will demonstrate how different number of thread will have effect on the computation time and CPU usage. Only computational time will be focused in this part and parameters are fixed.

By using parallel computing technique, multiple operations can be done at same time. Otherwise, it needs to iterate each operating process many times to finish the work. In this case, it saves time. However, the time that can be saved by parallel computing techniques is limited by number of processors that computer has. For example, the computer that conducts this experiment has 8 logical processor which means that the operation system can at most distribute resources to let 8 threads to do the computation at the same time. Any additional threads have to wait until any of these 8 threads finish its work. When this happens, the operation system will release the resources that previous thread occupy and dynamically distribute it to other waiting thread. It keeps working like this unit no more work needs to be done. From this example, we can see that the program does not ‘parallel’ all the time.

Parallel computing technique is used in two parts in this paper. The first part is to generate multiple initial solutions. Taking Iowa census county data for example, set iteration to 1000 and pool size to be 10, figure 18 shows the generation time with parallel and without parallel computing techniques.

|  |  |
| --- | --- |
| Type | Time(s) |
| Without parallel | 10.5433 |
| With parallel | 1.58217 |

***Figure 18: Initial 10 solutions generation time comparison between program without and with parallel computing technique. The time has almost been reduced to one tenth.***

Similar to initial generation time, recombined process time can also be reduced by using parallel computing but the process is a little bit more complicated. It needs to take care of the shared memory and communication between multiple threads.

The reason why we need to take care of shared memory and communication between multiple threads is that we only have one pool which can be considered as share memory but there are several threads running the program and they may want to write result back into pool at the same time. This will cause read after write (RAW) and write after write (WAW) hazard [14]. RAW means a data is read before new result writes into it and WAW means new result is written and overlaps the old result which previously wrote into the pool. Both conditions will cause program to get and process data that they are not supposed to process and generate wrong result. In order to avoid such problem, a parallel manager is used to monitor and manage all threads working. Whenever a thread finish working, it has to wait until all other thread finishing working before writes its result back into the pool. This waiting process thus takes extra time but it is essential to get results right (also called data consistence is hold).

This experiment uses 2, 4, 5, 10, 20 threads to run the recombined and GAT program to see the time spent and its CPU usage. Figure 19 shows the time and figure 20 shows CPU usage. The experiment is run on a computer with 8 processors.

From the table and graph, we can see that when the number of threads is less than 8,with thread increasing, corresponding time decreases. But the time decreasing becomes less obvious when thread number is greater than 8 due to processor limitation. The reason why time reduction is not exactly matched with threat increasing when thread number is less than 8 (for example, thread number is doubled from 2 to 4 but time reduction is less than half) is because parallel waiting and communication takes some extra time to hold data consistence.

|  |  |
| --- | --- |
| Thread | Time(s) |
| 2 | 13.3114 |
| 4 | 8.600157 |
| 5 | 7.01485 |
| 10 | 6.215379 |
| 20 | 5.761735 |

***Figure 19: Time consumption varies with the number of threads.***

***Figure 20: CPU usage based on different number of threads. When the number of threads is greater than 8, the CPU usage is always 100%.***

**Algorithm Performance Evaluation**

For a specific dataset, the final result will depends on a bunch of parameters. These parameters can be written as a set (see figure 21). In this part, how these parameters have effect on the final performance will be systematically analysed. Figure 22 shows all parameter values that will be used and the experiment will be based on the combination of all these parameters. This means that for each value in one type of parameter, every parameter in other types from the set has to be tested.

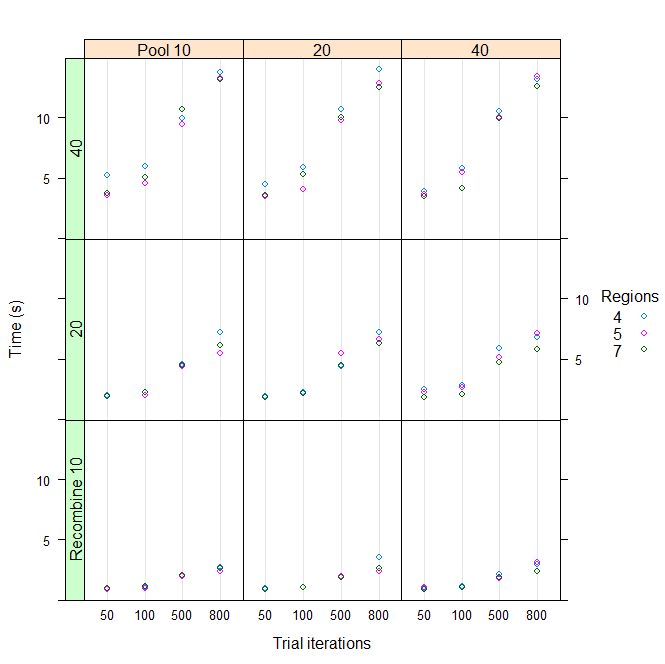
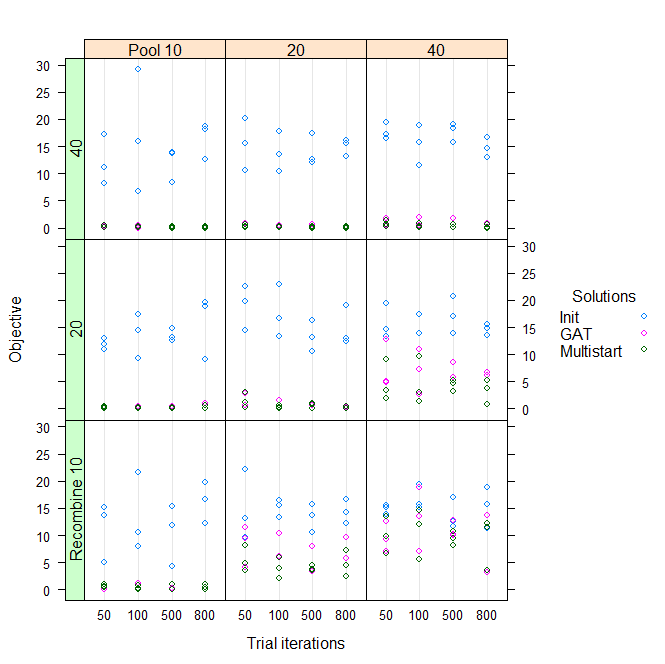
***Figure 21: Parameter set where RP is region part, RT is recombine time, PS is pool size and IT is iteration.***

|  |  |  |  |
| --- | --- | --- | --- |
| RP | RT | PS | IT |
| 4 | 10 | 10 | 50 |
| 5 | 20 | 20 | 100 |
| 7 | 40 | 40 | 500 |
|  |  |  | 800 |
| RP | **RT** | **PS** | **IT** |
| 5 | 10 | 10 | 50 |
| 10 | 20 | 20 | 100 |
| 20 | 40 | 40 | 500 |
|  |  |  | 800 |

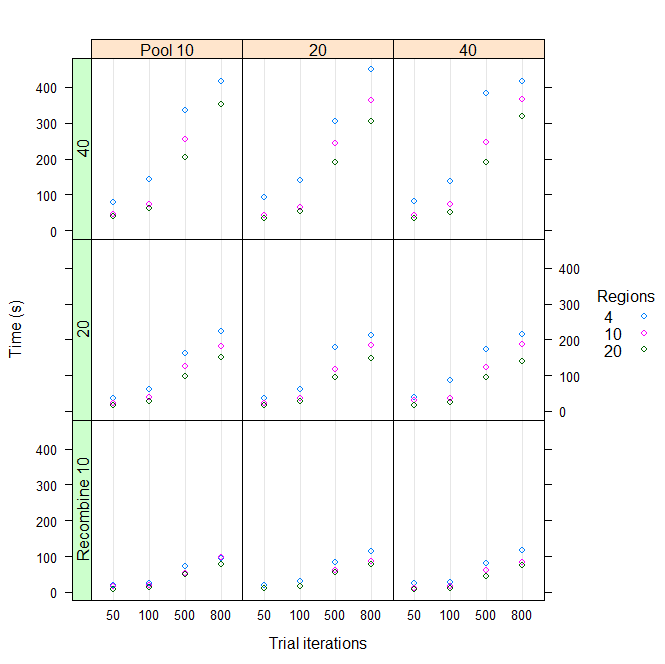
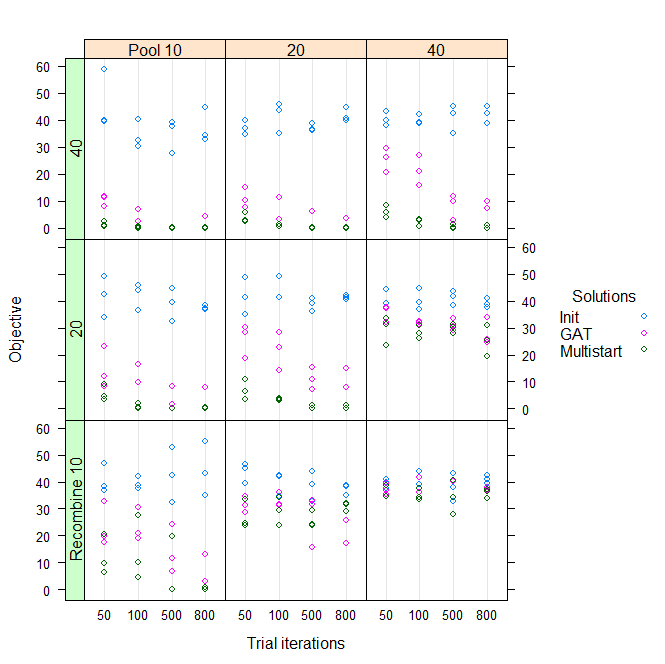
***Figure 22: Values for all the parameters that have effect on final performances. The experiments will based on all combination of these values. Upper table is census county parameters while lower table is census tract parameters.***

Figure 23 shows the overall performance and computation time of census county data. Figure 24 shows the overall performance and computation time of census tract data. Here the performance is based on the median objective function value from that pool and time is counted as second.

When the total number of iteration is low, the advantages of recombined method over pure GAT is not that obvious. That’s because the graph structure has not reached its local optimal point and there are still many suitable and valid swapping remaining for further improvement. This can be seen from the 50 iterations group where some pure GAT results are not always below recombined method results. Things are getting better when the total number of iteration increases, especially 800 iterations, when pure GAT cannot further improve its own result but recombined method can still improve the final result by taking advantages of mixture information from other solutions. Another issue is about the pool size and recombined time. When the pool size is relatively small, say 10, recombine times need not to be very high to ensure good performance. When pool size becomes larger, low recombine times cannot guarantee much low median value from the pool and sometimes it is even closer to the initial solutions. This becomes even obvious when the total number of iterations is also very low. This evidence can be seen from the graph when recombine times is only 10 and iteration is 50 but pool size is 40. This makes sense for the following reasons: first, low iterations does not make much improvement towards to results. Second, for each recombined process, two solutions are randomly picked from the pool to do the combined and GAT method. When the pool size becomes larger, low recombine time is not enough for many initial solutions to be picked and combined. In this case, there are some solutions that not picked to be improved by our method which will make median value of that pool relatively high, or to some extreme, close to initial value since that value is the one that has not been improved. But this situation is mitigated and ultimately disappeared with the increasing value of both recombine time and total number of iterations. What it shows on result is that pure GAT dots as well as recombined method dots become denser and they both have a large gap away from initial solution dots. Besides, dots from recombined method lays below dots from pure GAT. In terms of computational time, it is easy to see that more iterations, more recombine times lead to more computational time. But the number of new aggregated regions and pool size does not have much relationship with computational time. They only have effect on the final objective value.



***Figure 23:*** *performance and time of Census County*



***Figure 24:*** *performance and time of Census Tract*

**Summary and Discussion**

Spatial aggregation problem can be considered as a geographic optimization problem and how to use computational approach efficiently to solve this problem becomes critical in this area. Openshaw [15], Fotheringham [16], Armstrong [17]’s research has laid solid foundation in this area whose specification is what we called quantitative geography which is use computational and quantitative methods to solve geographic problem. Improving computational methods is much better than increasing computer hardware speed in terms of improving increasing efficiency. Exact method is not often used because it tries find all possible solutions which is very computational expensive. Heuristic method is considered as an effective way to solve geographic optimization problem though it cannot guarantee to reach optimal result every time.

Spatial aggregation problem can be considered as four types: selection problem without spatial constraints, selection problem with spatial constraints, partition problem without spatial constraints and partition problem without spatial constraints. Political aggregation problem is a partition problem with spatial constraints and GAT algorithm can be used to solve this problem. GAT is an efficient and effective heuristic method but it has two limitations. First, it cannot always generate good results and for those not that good results, its quality cannot be further improved when the total number of its iteration has reached to a certain. Another one is that it can only generate one result at a time. Recombining methods and parallel computing techniques are used in this paper to solve these two limitations. Results are analysis show that it works and produce satisfactory results.

This research also have some limitations. First, swapping and recombining process is purely random which may waste some computational time. In the future work, additional information may be added into the swapping and recombining process so that they will not be purely random. It can also decrease the useless number of iterations, thus further saving computational time. Besides, this paper only applies this improved method to population equality, but other factors like racial rate, income are also meaningful to be addressed. Further experiment may be done in a more border way in the future work.

**Acknowledgement**

I want to show special appreciation to my advisor, Ningchuan Xiao, who has provided me much help about this paper thinking and code writing. He also gives me many valuable comments in writing this paper.

[1] Martin, David. "Geography for the 2001 Census in England and Wales."*Population Trends* 108 (2002): 7-15.

[2] Kim, Myung Jin. "Optimization Approaches to Political Redistricting Problems." PhD diss., *The Ohio State University*, 2011.

[3] Assunção, Renato M., et al. "Efficient regionalization techniques for socio‐economic geographical units using minimum spanning trees." *International Journal of Geographical Information Science* 20.7 (2006): 797-811.

[4] Bacao, Fernando, Victor Lobo, and Marco Painho. "Applying genetic algorithms to zone design." *Soft Computing* 9.5 (2005): 341-348.

[5] Xiao, Ningchuan. "A unified conceptual framework for geographical optimization using evolutionary algorithms." *Annals of the Association of American Geographers* 98.4 (2008): 795-817.

[6] Densham, Paul J., and Gerard Rushton. "A more efficient heuristic for solving large p-median problems." *Papers in Regional Science* 71.3 (1992): 307-329.

[7] Alp, Osman, Erhan Erkut, and Zvi Drezner. "An efficient genetic algorithm for the p-median problem." *Annals of Operations research* 122.1-4 (2003): 21-42.

[8] Cova, Thomas J., and Richard L. Church. "Contiguity constraints for single‐region site search problems." *Geographical Analysis* 32.4 (2000): 306-329.

[9] Önal, Hayri, and Robert A. Briers. "Incorporating spatial criteria in optimum reserve network selection." *Proceedings of the Royal Society of London B: Biological Sciences* 269.1508 (2002): 2437-2441.

[10] Shirabe, Takeshi. "A model of contiguity for spatial unit allocation."*Geographical Analysis* 37.1 (2005): 2-16.

[11] Shirabe, Takeshi. "Districting modeling with exact contiguity constraints."*Environment and planning. B, Planning & design* 36.6 (2009): 1053.

[12] Bennett, David A., Ningchuan Xiao, and Marc P. Armstrong. "Exploring the geographic consequences of public policies using evolutionary algorithms."*Annals of the Association of American Geographers* 94.4 (2004): 827-847.

[13] Openshaw, Stan, and Liang Rao. "Algorithms for reengineering 1991 Census geography." *Environment and planning A* 27.3 (1995): 425-446.

[14] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative* approach. Elsevier, 2012.

[15] Openshaw, Stan. "Computational human geography: Towards a research agenda." (1994): 499-505.

[16] Fotheringham, A. Stewart. "Trends in quantitative methods II: stressing the computational." Progress in Human Geography 22.2 (1998): 283-292.

[17] Bennett, David A., Ningchuan Xiao, and Marc P. Armstrong. "Exploring the geographic consequences of public policies using evolutionary algorithms."Annals of the Association of American Geographers 94.4 (2004): 827-847.