Using Genetic Algorithms to solve the Minimum Labeling Spanning Tree Problem

Oliver Rourke, oliverr@math.umd.edu
Supervisor: Dr B. Golden, bgolden@rhsmith.umd.edu
R. H. Smith School of Business

May 11, 2012

Abstract

Genetic Algorithms (GAs) have been shown to be very powerful tools for a variety of combinatorial optimization problems. Through this project I have implemented a GA to solve the Minimum Labeling Spanning Tree (MLST) problem (a combinatorial optimization problem). I extended earlier work in this area both by improving upon the serial algorithm and by developing a parallel implementation of the GA, which involved designing and testing various inter-processor communication schemes. The resulting parallel GA was tested on a database of problems and compared both with the original GA and with other heuristics from the literature.

1 Background and Introduction

1.1 Problem: The Minimum Labeled Spanning Tree

The Minimum Labeling Spanning Tree (MLST) was first proposed in 1996 by Chang and Leu [4] as a variant on the Minimum Weight Spanning Tree problem. In it we are given a connected graph G (composed of edges, E, and vertices, V). Each edge is given one label (not necessarily unique) from the set L. I denote $|E| = e$, $|V| = v$ and $|L| = l$. One such graph is shown in Figure 1. A sub-graph is generated by using only the edges from a subset $C \subset L$. The aim of the problem is to find the smallest possible set of labels which will generate a connected subgraph. More than one global minimum (equally small sets) may exist, although we are satisfied if we identify one. Real world applications include the design of telecommunication [12] and computer networks [15].

This problem has been shown to be NP-Complete [4], and we therefore must use a heuristic to obtain near-optimal results in a reasonable amount of time (guaranteeing that this is the true optimum solution will take unreasonably long). In this paper, a solution is a set of labels, and we will call a set 'feasible' if the sub-graph generated by the set of labels is connected.

1.2 Existing (non-GA) Heuristics

Several heuristics have been proposed to solve the MLST problem. The first heuristic, the Maximum Vertex Covering Algorithm (MVCA), was proposed by Chang and Leu [4] (with correction by Krumke [10]) and runs in polynomial time. Other heuristics that have been used include Simulated Annealing, Tabu Search, Pilot Method, Greedy Randomized Adaptive Search and Variable Neighborhood Search [3, 6]. Comparisons of these methods from the literature [3, 6] suggest that the Variable Neighborhood Search (VNS) returns the best quality of solution. Both the MVCA and VNS were encoded to give a benchmark (with descriptions included in Appendix B).

2 Genetic Algorithms - Theory

Genetic algorithms (GAs) are a class of heuristic which have been widely used to solve combinatorial optimization problems (see Dorronsoro and Alba [7] for an extensive review). These algorithms apply the Darwinian notion of natural selection on a set of feasible solutions (with each solution composed of a number of 'genes'), iterating through successively 'stronger' generations. By modeling many different possible solutions, we hope to be able to investigate a large portion of the solution space, and by carefully choosing the interactions between these solutions we aim to select the strongest 'genes' to include in the next generation. This process can be broken down into six key steps.

2.1 Key steps in a Genetic Algorithm (GA)

2.1.1 Initialization

The first step in a genetic algorithm is to create an initial generation of feasible (valid) and varied solutions. At this stage we are not concerned with their 'fitness'.

2.1.2 Selection

The next step is to choose pairs of individuals ('parents') to be bred from the current population. Simpler strategies include iterating through all possible combinations and randomly choosing sets of parents. Goldberg [9] and Collins et al. [5] compare a variety of more complex strategies, including Linear Rank Selection, Proportionate Reproduction, Tournament and Genitor's Selection. These have been devised to favor breeding 'better' pairs, but to nevertheless occasionally breed 'inferior' pairs (to maintain the genetic diversity).

2.1.3 Combination

The combination operator mixes genetic material from the two parents create a feasible offspring solution (containing some genetic material from each of the parents). Some GAs try to pick the 'strongest' genes from each of the parents [16], while others randomly combine genes to create a feasible offspring [11]. The first strategy should converge faster, but is also more likely to converge to a point away from the global minimum.

2.1.4 Mutation

The mutation operator creates new genetic material in the offspring. This is done by modifying the offspring in a random manner. In the literature this operator is applied very rarely ($\approx 1 - 5\%$ of the time) as it often makes the offspring less competitive [7].

2.1.5 Replacement

The final step is replacement. The next generation is created by choosing the strongest individuals from the set of offspring and parents ('survival of the fittest'). The algorithm then loops back to Step 2 (Selection), with evolution now happening on the new, hopefully better, population.

2.1.6 Termination

A termination condition determines when we leave the evolution loop and return the best individual as a result. This can either be pre-determined (such as a set number of generations/amount of time) or it can depend on the state of the population (the population has stagnated/we have an 'acceptable' solution).

3 An existing Genetic Algorithm for the MLST

In 2005, Xiong et al. [16] implemented a GA to solve the MLST. Each individual in the population is represented by a set of labels. A solution is feasible if the sub-graph it generates is connected, and the strength of a solution is given by the number of labels in the set (with fewer labels corresponding to a stronger solution). This GA was devised to be simple having only one parameter and requiring no fine-tuning. I will denote the set of all individuals (the entire population) as P, with size $|P| = p$ constant over all generations.

3.1 Steps in Xiong's GA

1 - Initialization: Each individual starts off as an empty set (not feasible). Labels are randomly added (without duplication) until the individual becomes feasible.
2 - Selection: The jth offspring will be formed by breeding parents enumerated j and $(j + k) \bmod p$, where k is the generation number. This sweeping pattern allows every individual to breed with every other individual in turn.
3 - Combination: Pseudo-code for the combination algorithm is included in Appendix B, and the method is shown diagrammatically in Figure 2a. This algorithm considers all labels inherited from both parents, and favors those labels which appear most frequently in G (under the assumption that more frequent labels will be more useful to the offspring).
4 - Mutation: Pseudo-code for the mutation operator is likewise included in Appendix B and the technique is demonstrated in Figure 2b. This works by adding a random label to the offspring's set of labels, and then one by one attempting to remove labels from the set (starting with the least frequent label in G), discarding labels where the resulting solution is feasible. This operator will be applied to all offspring. Note that this generates viable offspring that do not contain any excess labels (no label can be removed while keeping the offspring feasible).
5 - Replacement: The jth offspring will replace the jth parent if it is 'stronger' than the parent.
6 - Termination: The algorithm stops after p generations.

3.2 Running Time Analysis

Given a set of labels $C \subseteq L$, viability can be determined by a depth-first search (DFS) in $O(e+v)$ operations. This will happen a maximum of $2l$ times in each instance of breeding (l times in combination, l times in mutation), and there are p^2 instances of breeding (p generations with p breeding pairs in each generation). Finally note that in any connected graph $v - 1 \leq e \leq v(v - 1)/2$, but in most test instances I will use $e = O(v^2/2)$. Therefore the upper bound on the operation count is $O(lp^2v^2)$.

3.3 A variant on Xiong's GA

In a later paper, Xiong et al. [17] proposed a more computationally intensive genetic algorithm known as the Modified Genetic Algorithm (MGA). This was shown to outperform the original GA. The difference between the GA and the MGA is in the combination operator - essentially the MGA performed the MVCA algorithm, starting with the union of genes from both parents. An outline of this new combination operator is included in Appendix B. The MGA is currently the strongest genetic algorithm for the MLST in the literature.

4 Modifications to the Serial GA

The first part of my project was to attempt to improve the serial Genetic Algorithm proposed by Xiong et al. [16,17] while keeping the algorithm running in serial. It was observed that the population in the algorithm often prematurely converges to a non-optimal solution. This may be avoided by increasing the diversity in the population; accordingly the combination, mutation and replacement operators were modified to cause and maintain a greater diversity in the population.

4.1 Stochastic Operators

One way in which diversity might be better promoted in the algorithm is to modify the crossover and mutation algorithms to make them more stochastic. Xiong's algorithm operators always favored the same genes, those which appear most frequently in the graph. Although it might be possible to perform more complicated analysis initially to come up with a more effective technique, I hoped to come up with a simple technique which merely increased diversity and let the selection operator decide when the new operator had worked.

This was done by randomly modifying the list in which genes are added or removed from the graph every time the list is used. When iterating through the list (using the same order in the GA) I try skipping the next gene (label) with a pre-determined probability p. Note that the algorithms must now be modified so that it can continuously loop over the list until either a valid individual has been created (crossover operator) or until an attempt has been made to remove each gene (mutation operator). The probability was set to $p = 50\%$ before any tests were run to stay true to the original GA's goal of remaining competitive without fine tuning.

4.2 Keep equal offspring

A second method to promote diversity in the population is to modify the selection operator to encourage greater variation. One simple way of doing this is to replace the parents with the offspring if the offspring are fitter than or equally fit as the parents (as opposed to only replacing if fitter). It was hoped that this tactic would lead to less chance of the population getting stuck away from the global minimum.

4.3 Force Mutation

One final method to encourage greater diversity in the population is to modify the mutation operator to favor retention of the new, randomly selected gene. In the original algorithm the new gene is treated the same as all the others - this means that if it occurs less frequently in the graph than the inherited labels it will always be removed first (the inherited labels form a viable solution) and it will never actually have been tested. To avoid this the mutation operator was modified such that the new gene is marked and I do not attempt to remove it until I have tried to remove all other genes. This may involve looping several times through inherited genes if a mutation coin toss is also in use.

5 Designing Parallel Code

The second part of the project was to develop a parallel implementation of the GA. The main reason for such a parallel algorithm is speed. By using a large number of processors running at or near capacity I expect better results in less clock time. This would also allow us to run larger problem instances which would take too long if only run on a single processor.

5.1 Parallel Architecture

Parallel algorithms can be broadly classified into one of two categories depending how processors communicate with each other.

Under a **Master-Slave** classification one processor (the 'master') is in control of the entire heuristic. This involves issuing commands to the other processors (the 'slaves'), receiving/interpreting results and issuing new sets of commands dependent on all the information received. This centralized configuration is relatively easy to implement, and has been widely used in combinatorial optimization problems [2, 11, 14]. However it does not scale well. When run on large arrays of processors a bottleneck forms around the 'master' as it is unable to process all the information and keep up with demand.

The alternative is to implement a scheme which uses **direct communication** between the processors. In such a scheme, information about the state of each processor is sent directly to the other processors, with each processor modifying its search in light of information received. This architecture has been shown to scale very well [8], allowing it to return strong results from very large arrays of processors.

5.2 Synchronous vs. Asynchronous code

Parallel algorithms may also be classified according to when the processors communicate with each other. A comparison of the two techniques to solve a combinatorial optimization problem was carried out by Barbucha [1].

Synchronous code ensures that all processors are working in time with each other. Information is only shared at certain pre-arranged points in the code, making the inter-processor communication quite straightforward. But this is also its weakness: faster processors will be restrained to working at the pace of the slower processors. This means the entire process will not run at optimal speed.

A better alternative is **asynchronous code**, in which all processors are allowed to operate at their own maximal pace. In turn this means that communication between processors may occur at any point in the algorithm. This is more difficult to design but it should operate at higher speeds.

5.3 Shared memory vs message passing

Communication between processors can be divided into two classes.

Parallel algorithms can be constructed using **Message passing** to communicate between processors (often, but not necessarily, governed by the Message Passing Interface (MPI) protocol). Such parallel algorithms are relatively simple to design, as information can only be passed between processors in certain pre-arranged formats and at certain pre-arranged times. Moreover, algorithms which use message passing can be run on any grid with connected processors (there is no need for access to shared memory).

In contrast, parallel algorithms which are to be run on grids with **shared memory** may communicate by altering the shared variables. This requires implementing locks (such as mutexes and condition variables) to ensure multiple processors do not try to simultaneously access shared variables. Communication via shared memory is much more flexible than message passing, allowing both a wide variety of objects to be shared and allowing processors to access shared memory at any point in time (assuming it is not temporarily locked by another processor). This flexibility allows for completely asynchronous code, which will best use all processor power available at any given moment.

For this project I have designed GA code which uses Pthreads (shared memory) with direct communication between processors. This code exists in both asynchronous and synchronous forms. The synchronous code has some limited coordination between processors so that all processors will start each generation simultaneously, but all calculations within each generation are completely asynchronous. The parallel VNS code is completely asynchronous with no inter-processor communication (beyond initialization of threads). This means that the each processor performing VNS runs independently of all others, returning its result when the limit (computational time or iteration count) has been reached.

6 Parallel Genetic Algorithm code for the MLST

Next I considered a parallel implementation of a genetic algorithm. In designing such code I had to consider both how the computational effort was to be divided amongst processors and how the processors might be able to communicate with each other to achieve the strongest results.

6.1 Imposing a Population structure

The simplest way to split a genetic algorithm over multiple processors is to divide the total population into smaller groups which can be allocated to the various processors. Ideally the majority of interaction will happen between individuals on the same processor, limiting inter-processor communication. Three different population arrangements are discussed below (for more information see Alba&Dorronsoro [7]). Diagrams of each arrangement are included in Figure 3, although for this project I primarily concerned myself with Distributed Genetic Algorithms

6.1.1 Panmictic

A Panmictic GA allows all individuals to breed with all other individuals, like Xiong's original genetic algorithm. This type of genetic algorithm does not parallelize well as individuals are as likely to interact with individuals across the whole population, requiring much inter-processor communication when the population is split across several processors.

6.1.2 Distributed

Distributed GAs impose an island structure on the population, in which each individual belongs to an island and is only able to breed with other individuals which belong to that island. Genetic material is carried from one island to the next only very slowly, through a migration operator. This scheme is the easiest to parallelize, with subpopulations equally distributed over available processors and inter-processor communication is only needed for the migration operator. This is the structure I used for my parallel GA.

6.1.3 Cellular Genetic Algorithm

Cellular GAs use a mesh structure on the population. Each individual is located at some node in the mesh and is only able to breed with other individuals within a local neighborhood on the mesh. The key point here is that the neighborhood is normally much smaller than the total population. Cellular GAs can be converted so that they run in parallel so by allocating different parts of the mesh to different processors (so that the only inter-processor communication required is for those individuals whose neighborhood runs over multiple parts/processors). In general these GAs require more inter-processor communication and are more difficult to code than the above Distributed GAs.

6.2 Migration Operators

When working with a Distributed GA it is possible to modify the migration algorithm to achieve more useful inter-processor communication. I considered three different types of migration operators.

6.2.1 No Migration

The first migration scheme I shall consider is one in which there is no communication between the subpopulations. This is the simplest scheme, and will be used as the benchmark to determine how well other communication schemes have performed.

6.2.2 Local Migration

Another possible migration scheme was proposed by Scharrenbroich [13]. Under this scheme the subpopulations are arranged on a toroidal mesh with a local neighborhood defined around each subpopulation. Individuals may now only migrate to a subpopulation from its neighboring subpopulations. This concept is illustrated in Figure 4. In Scharrenbroich's genetic algorithm migration occurs when a subpopulation has stagnated (after K generations with no improvement), involving copying each of its neighbor's best solutions into its own subpopulation (replacing its worst solutions).

The above migration scheme replaces the weakest individual in each population, reducing diversity especially for small subpopulations. To avoid this I formulated a variant on Scharrenbroich's local migration scheme which does not involve replacing individuals (the 'Waiting Room' approach). Under this scheme the best individuals from each subpopulation are placed in shared memory and updated when a better individual is found in the respective subpopulation. To create a breeding pair each subpopulation may now pick from any of its own individuals or those saved in the shared memory.

6.2.3 Global Migration

An alternative migration scheme involves all the subpopulations communicating indirectly with each other through some shared location in memory, the 'vault'. I constructed a scheme such that this shared memory location might be used to record the best $vault_{pop}$ individuals found on any processor to date. This means that after every breeding instance the offspring solution is compared to the individuals in the vault, and if found to be better than any of them (and different from all of them) it replaces the weakest individual in the vault. There is an additional parameter, $vault_{breed}$, which controls the extent to which individuals in the vault interact with individuals in the subpopulations. In every breeding instance there is a probability, given by $vault_{breed}$, that Parent2 will be replaced with a randomly selected individual form the vault. In this way the strongest individuals found to date are shared amongst all the subpopulations. Note that for all trials below I chose $vault_{pop} = 20$, $vault_{breed} = 20\%$

The construction of the vault has benefits beyond a more centralized communication scheme. As the vault maintains the best individuals separate from the subpopulations, it is now guaranteed that the best solution found to date will be kept in the vault. This in turn means that each subpopulation no longer needs to keep stronger individuals. The most successful methods I found to take advantage of this were individual and population 're-spawn'. With individual re-spawn there is a probability (set at 20% for all results below) that an individual will be reinitialized by random after each breeding instance. Under population re-spawn each subpopulation of individuals will be completely reinitialized at random if no improvement has been achieved after a set amount of time (set at 5 generations for all results below).

7 Implementation, Hardware

All heuristics will be encoded in C++ with POSIX threads (Pthreads) used for inter-processor communication. All code was run on the Genome6 cluster at the University of Maryland, consisting of 8 Quad-core AMD Opteron® Processors (2.6GHz).

8 Databases

The original paper by Xiong et al. [16] outlined a method to randomly generate connected, labeled graphs with a variable number of vertices, colors and edges. This technique was later used by Cerulli et al. [3] to generate 135 sets of instances (with 10 graphs per set) with various combinations of the key parameters. These sets of instances are useful for validation, however the majority of the sets used involve smaller graphs which are relatively easier to solve. I also found that having only 10 graphs per set did not allow for a thorough investigation of the efficacy of heuristics, as often all heuristics would report very similar scores with perhaps one more/fewer label used by a heuristic on one of the instances. Therefore I used the same technique from Xiong [16] to create my own sets of instances. I generated 5 sets of instances, each with 100 nodes, 0.2 edge density and either 25, 50, 100, 250 or 500 labels.

9 Validation and Testing Technique

9.1 Validation of Serial Code

In order to validate my GA code I have obtained a copy of the original Genetic Algorithm code by Xiong. By initializing the random number generators in the same way and ensuring that the random number generators are called in precisely the same order I was able to ensure that exactly the same results are returned from both sets of computer code. This was done both by running both sets of code on exactly the same instance, verifying after every generation that the population was the same, and by running on 200 diverse instances and verifying that both returned the same set of labels (from the best individual in the population).

To validate my VNS code it was run on the same sets of graphs as used in Consoli [6] (where the most successful version of the VNS for the MLST was first proposed), and results compared. This was found to be the case (within experimental error due to unknown random number seed).

9.2 Validation of Parallel Code

To validate the parallel code I initially ran the code with no communication between the processors (other than initializing each subpopulation and reporting back of the best solution) for two separate sets of instances. I then compared the results, verifying that each processor returned a solution in keeping with what was expected from the serial runs (to within experimental error).

The second step in parallel validation was to ensure that the migration operators and inter-processor communication were both acting as expected. This was done by flagging certain individuals within the population (via the inclusion of a unique gene) and watching them be copied between processors (and the vault if appropriate) and verifying by hand that all movements were as expected/scheduled. This was found to be the case.

The final step in validating the parallel implementation was to compare the running time for the parallel code with the running time for the serial code. I then modified the code to investigate the delay caused by synchronizing the processors (so that all processors have start a new generation together), and the delays due to more intense communication between processors by including a vault. This was done on two of the sets of instances I generated, those with $L = 100$ and $L = 500$, with the time required for each of these runs in Table 2 in Appendix C. This shows the parallel code ran at an efficiency of about 75% for no communication/synchronization (delay cause by overhead from initializing, running and closing the multiple threads), and about 50% for full synchronization/communication (additional delays due to synchronization between processors and inter-processor communication from vault/individual re-spawn).

9.3 Testing the Serial Implementation

The serial heuristics was then run over my test instances (see 'Databases' above) to test performance both in terms of results and computational time. For stopping conditions based off iteration count I used 20 iterations for the GAs (same as Xiong [16] and 20 iterations of VNS (as this was found to be comparable to 20 GA generations in computational time for many instances). For stopping conditions based off computational time I allowed $T = 20 * L$ms per instance for both GAs and the VNS. Such limits are not relevant to the MVCA which runs (and finishes) in polynomial time. All algorithms will be run 10 times with different random number seeds, recording both the average number of labels required per instance and average time per instance (when stopping conditions is dictated by iteration count).

9.4 Testing the Parallel Implementation

The parallel algorithm will be tested by running over my test instances to test performance. The same stopping conditions will be used as the serial heuristic, with time being measured across all processors for the computational time limit stopping condition and all processes being terminated once the time limit has passed. Generally parallel testing will be done using 32 processors, although I did do some testing with 8 processors for the sake of comparison.

10 Results and Discussion

10.1 Serial Heuristics

Testing The MVCA, the VNS and the various versions of Xiong's GA and MGA were run as described in Section 9 with constant iteration count. Results are recorded in Table 1. Three of the changes proposed in Section 4 led to a significant improvement in the computational results for the standard GA (all except "Keep Equal"), with at most a small increase in running time in all cases. When all the changes were used together, greater improvements yet were typically achieved. When the new operators (excluding Stochastic Crossover) were used on the MGA the results returned were always slightly stronger than when the original operators were used, with simulations generally taking slightly less time. Over most of the sets of instances the GA with the new operators appears to perform at a comparable level to the MGA but in much less computational time. For almost all sets of instances the GAs appear preferable over the VNS in both solution quality and running time.

Figure 5 records serial runs with a stopping condition for constant computational time (and where the code has been modified to return results every 10 milliseconds). This gives a clearer picture as to how the algorithms are performing. Xiong's original GA is converging the fastest, performing the best over very short periods but apparently stagnating away from the optimal solution. It is then passed by the GA with stochastic operators, which is initially hindered by the stochastic approach but stagnates later and significantly outperforms the original GA. Both the original MGA and the MGA with stochastic operators converge slower than either of the GAs, passing the original GA and approaching a solution quality similar to (although in all cases definitely worse than) the GA with stochastic operators. For the sets with fewer labels, the VNS appears to converge slower but to solutions of comparable quality when compared with the best GAs, but its rate of convergence slows dramatically for sets with more labels (rendering it greatly inferior to al GAs).

These results appear to support the idea that changing operators in order to obtain greater diversity in the population will generally give a better and more robust GA.

10.2 Parallel Heuristics

Table 3 contains data gathered when the distributed algorithms were run again using the migration operators outlined above in Section 6 for the set of instances with 250 labels (chosen to be relatively difficult to solve), using either 8 or 32 processors. The One Sided Mann-Whitney U-Test was used to test if the migration schemes led to a significant change in the performance of the heuristics.

In three of the four test cases the local migration operators did not lead to a significant improvement over no migration ($\alpha = 0.05$, the exception being with the waiting room approach with 32 processors). In all cases, the waiting room approach apparently outperforms the replacement strategy, suggesting the importance of maintaining diversity within subpopulations. This contrasts with the global migration approach, which yielded a significant improvement over no migration in all cases tested ($\alpha = 0.05$). Moreover the individual re-spawn term always yields stronger results than the plain vault with no migration (although the vault re-spawn term appears to weaken the vault migration in this case). These results validate the use of global migration, but also suggest that some gains might be had in using local migration if care is taken.

Figure 6 records the results from the parallel heuristics when run on all of my sets of instances using 32 processors with a time limit stopping condition. For this I only considered the more successful global migration approach. The results for the GA here broadly agree with those discussed above (and recorded in Table 6), with the vault and individual re-spawn significantly outperforming the GA with no communication while the vault with population re-spawn narrowly beats the no communication approach (perhaps performing weaker than expected). In all cases it can be seen that the MGA and the GA with stochastic operators (here labelled GA2) are once again the most successful heuristics. These techniques do not appear to benefit from the vault migration approach - indeed, in many cases the re-spawn tactics cause the heuristics to converge slower. This means that the solution quality at the end of the allotted time is worse than that attained when no migration was implemented. This may have changed had the heuristics been run for longer amounts of time. Finally the VNS algorithm performs at a comparable level to the GAs for the sets with small numbers of labels, but is comparatively weaker for the sets with more labels.

These results show that the implementation of well-designed migration operators in a parallel algorithm may greatly help in some cases, although the schemes investigated do not appear to work for all GAs over all instances (for the time frames chosen).

11 Schedule

- **Create my serial GA - Done**
  Tasks: Modify Xiong's code, build sets of graphs for testing
  Dates: October 2011
  Result: Competitive, efficient GA code
  Validation: Compare with other heuristics and global minima (when known)

- **Convert to a parallel GA - Done**
  Tasks: Modify above GA, initially to synchronous and later asynchronous parallel code (using direct communication between processors)
  Dates: November-December 2011
  Result: Both asynchronous and synchronous versions of efficient, parallel GA code with direct communication
  Validation: Compare results and speedup against serial code

- **Fine tuning parallel GA - Done**
  Tasks: Experiment with different population arrangements, migration operators to obtain optimal parallel GA. Possibly design and implement a Cellular GA for comparison (this part not done)

Dates: January-February 2012
Result: Competitive versions of earlier GA code
Validation: Compare results and speed with earlier versions of parallel code and other heuristics

- **Large-scale testing, presentation - Done**
  Tasks: Run optimized code on large array of processors (GENOME cluster at UMD), analyze all results, prepare report and presentation
  Dates: March-May 2012
  Result: Final results and analysis of the whole GA in the form of a formal report and presentation.
- **Extra - Not Done**: Time permitting, modify GA code and test on a different combinatorial optimization problem.

12 Deliverables

- Competitive, efficient serial and parallel code for a GA on the MLST problem using direct processor-processor communication in both asynchronous and synchronous form.
- Results from tests on various different sized arrays of processors and across various problem instances.
- Report(s) and presentation(s).

References

[1] Dariusz Barbucha. Synchronous vs. Asynchronous Cooperative Approach to Solving the Vehicle Routing Problem. *ICCCI*, LNAI 6421:403–412, 2010.

[2] Jean Berger and Mohamed Barkaoui. A parallel hybrid genetic algorithm for the vehicle routing problem with time windows. *Computers & Operations Research*, 31:2037–2053, 2004.

[3] Raffaele Cerulli, Andreas Fink, Monica Gentili, and Stefan Voss. Extensions of the minimum labelling spanning tree problem. *Journal of Telecommunications and Information Technology*, 4:39–45, 2006.

[4] Ruay-Shiung Chang and Shing-Jiuan Leu. The minimum labeling spanning trees. *Information Processing Letters*, 63:277–282, 1997.

[5] Robert Collins and David Jefferson. Selection in Massively Parallel Genetic Algorithms. *Proc. of the Fourth International Conference on Genetic Algorithms*, pages 249–256, November 2007.

[6] S Consoli, K Darby-Dowman, N Mladenović, and J A Moreno Pérez. Greedy Randomized Adaptive Search and Variable Neighbourhood Search for the minimum labelling spanning tree problem. *European Journal of Operational Research*, 196:440–449, 2009.

[7] Bernabe Dorronsoro and Enrique Alba. *Cellular Genetic Algorithms.* OPERATIONS RESEARCH/COMPUTER SCIENCE INTERFACES. Springer, New York, 2008.

[8] Lucia M A Drummond, Luiz S Ochi, and Dalessandro S Vianna. An asynchronous parallel metaheuristic for the period vehicle routing problem. *Future Generation Computer Systems*, 17:379–386, 2001.

[9] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms.* Morgan Kaufmann, San Diego, 1991.

[10] Sven O Krumke and Hans-Christoph Wirth. On the minimum label spanning tree problem. *Information Processing Letters*, pages 81–85, 1998.

[11] Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31:1985–2002, 2004.

[12] S Raghavan and G. Anandalingam. *Telecommunications Network Design and Management.* . New York: Springer., 2003.

[13] Max Scharrenbroich and Bruce Golden. A Parallel Architecture for the Generalized Travelling Salesman Problem: Final Report. Technical report, 2009.

[14] A Subramanian, L M A Drummond, C Bentes, L S Ochi, and R Farias. A parallel heuristic for the Vehicle Routing Problem with Simultaneous Pickup and Delivery. *Computers and Operation Research*, 37:1899–1911, 2010.

[15] Andrew S Tanenbaum. *Computer Networks.* Prentice Hall, Upper Saddle River, 2003.

[16] Y Xiong, B Golden, and E Wasil. A One-Parameter Genetic Algorithm for the Minimum Labeling Spanning Tree Problem. *IEEE Transactions on Evolutionary Computation*, 9(1):55–60, February 2005.

[17] Yupei Xiong, Bruce Golden, and Edward Wasil. Improved Heuristics for the Minimum Label Spanning Tree Problem. *IEEE Transactions on Evolutionary Computation*, 10(6):700–703, December 2006.

Appendix A: Diagrams

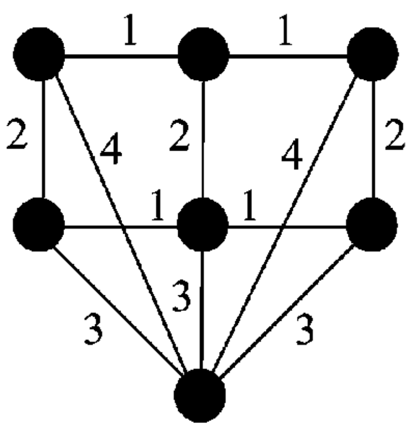
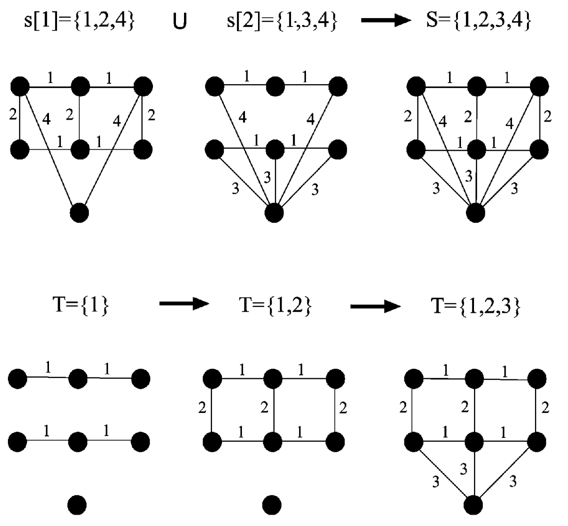
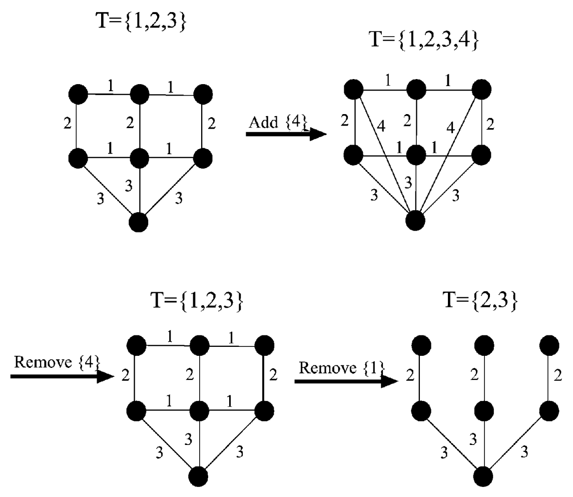


Figure 1: An example of a labeled spanning tree [from [16]]. In this case, the (unique) minimum set of colors which will generate a connected sub-graph is {2, 3}



(a) a



(b) a

Figure 2: Breeding operators from Xiong's GA [16]

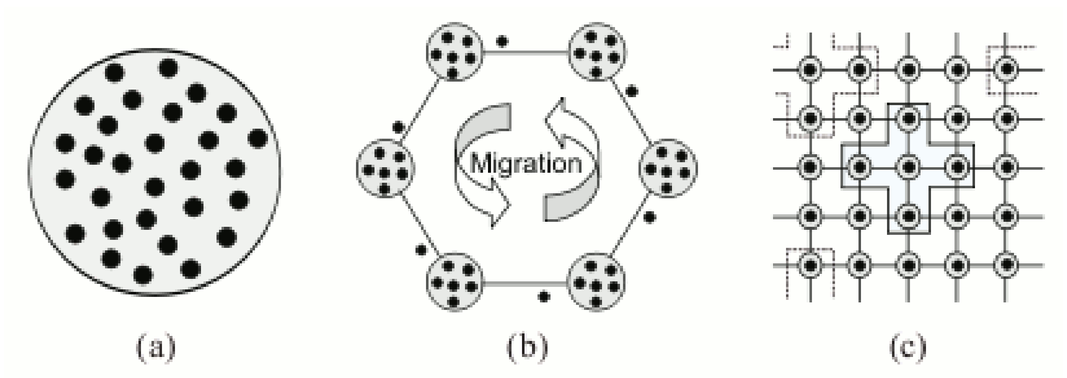


Figure 3: Three different population arrangements - a) panmictic, b) distributed, c) cellular genetic (CGA) [from [7]]. In each of these a dot represents an individual (solution) which can 'breed' with other solutions within its shaded neighborhood.

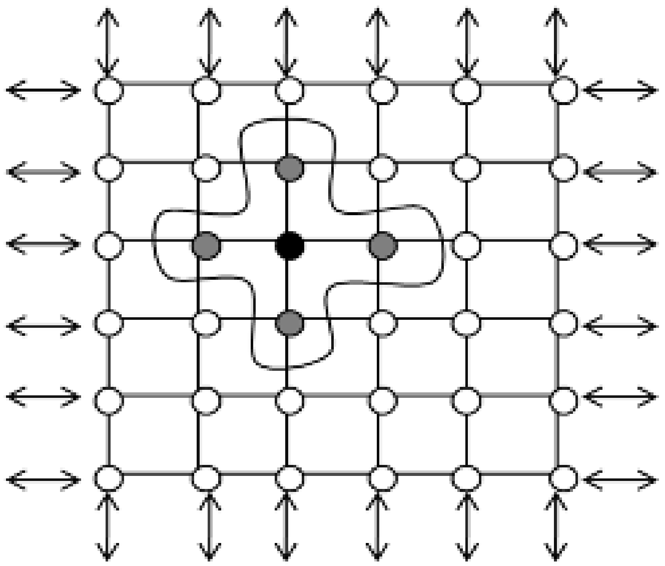


Figure 4: The Distributed Genetic Algorithm implemented by Scharrenbroich. Each 'node' now represents a processor, monitoring its own subpopulation, with the arrows representing migration between subpopulations [13]

Appendix B: Algorithms

Note: G denotes the graph, composed of vertices V and edges E, each edge having a label belonging to the set of labels L.

The MVCA algorithm

```
Let $C = \emptyset$, $H$ is the subgraph induced by $C$ (updated whenever $C$ is changed)
while $H$ is not connected do
    for $i \in L - C$ do
        Add $i$ to $C$
        Count the number of connected components in $H$
        Remove $i$ from $C$
    end for
    Add the color which resulted in the least number of connected components
end while
```

The VNS algorithm

```
$S$ = Generate-Initial-Solution-at-Random
repeat
    $k = 1$, $k_{max} = |S| * 4./3$
    while $k < k_{max}$ do
        $C'$ = Shaking($C$, $k$)
        LocalSearch($C'$)
        if $|C'| < |C|$ then
            $k = 1$, $C' \to C$, $k_{max} = |C| * 4./3$
        else
            $k = k + 1$
        end if
    end while
until terminating-condition

function SHAKING($C$, $k$)
    for $i = 1 \to k$ do
        $r$ = random(0, 1)
        if $r < 0.5$ then
            Add random label from $L - C$ to $C$
        else
            Remove random label from $C$
        end if
    end for
    return $C$
end function

function LOCALSEARCH($C$)
    if $C$ is not feasible then
        Apply MVCA to $C$
    end if
    Try to remove each label from $C$ such that $C$ remains feasible
end function
```

Outline of Xiong's GA

```
for Individual in Population do
    Generate-Initial-Solution-at-Random
end for

for $g = 1 \to p$ do
    for $i = 1 \to p$ do
        $S1$ = Individual[$i$]; $S2$ = Individual[$(i + gen) \bmod p$]
        offspring = crossover(S1, S2)
        offspring = Mutate(Offspring)
        if offspring is better than parent1 then:
            Individual[$i$] = offspring
        end if
    end for
end for
```

Crossover Operator from Xiong's GA

```
function Crossover($S1$, $S2$)
    $S = S1 \cup S2$, $T = \emptyset$
    Sort $S$ in decreasing order of frequency of labels in $G$.
    Add labels of $S$, from first to last, to $T$ until $T$ is feasible
    return $T$
end function
```

Mutation Operator from Xiong's GA

```
function Mutate($S$)
    Randomly select $c \in L - S$
    $S = S \cup c$
    Sort S in decreasing order of frequency of labels in G
    From last to first, try to remove labels from S such that S remains feasible
    return S
end function
```

Combination operator from Xiong's MGA

```
function MGACrossover($S1$, $S2$)
    Let $S = S1 \cup S2$
    Apply MVCA to $S$
    return $S$
end function
```

Appendix C: Results

Key to Tables:
GA- variants of Xiong's Genetic Algorithm [16] :
SC = Stochastic Crossover, SM = Stochastic Mutation, KE = Keep Equal, FM = Force Mutation, All = SC+SM+KE+FM.
GA+SM+SC is also referred to by GA Stochastic or GA2
MGA- Xiongs Modified Genetic Algorithm [17] MGA Stochastic = MGA+SM Non-GA algorithms:
MVCA = Maximum Vertex Covering Algorithm from Chang and Leu, 1997 [4]
VNS = Variable Neighbourhood Search from Consoli et al. [6]

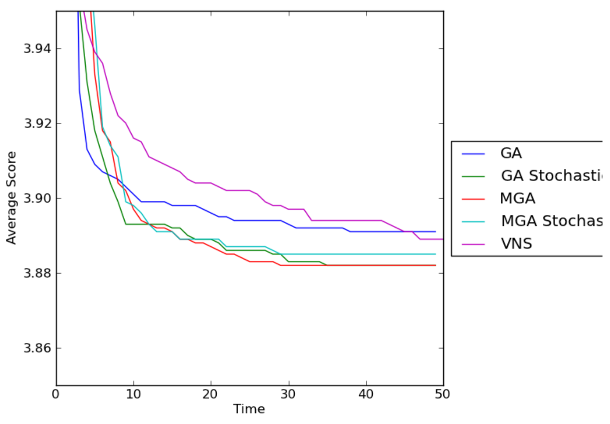
Parameters			Non-GA Heuristics		Original GA + variants						MGA + variants	
n	l	d	MVCA	VNS	Original	+SC	+SM	+KE	+FM	+All	Original	+All
25			4.000	3.91	3.894	3.882	3.881	3.911	3.883	3.883	3.884	3.882
50			6.090	5.87	5.896	5.831	5.830	5.931	5.802	5.778	5.820	5.785
100			9.440	9.00	8.906	8.710	8.784	8.993	8.753	8.669	8.716	8.714
250			15.500	14.96	14.784	14.535	14.586	14.925	14.640	14.477	14.446	14.434
500			22.170	21.38	20.916	20.606	20.656	21.016	20.804	20.598	20.557	20.546

(a) Average number of labels required

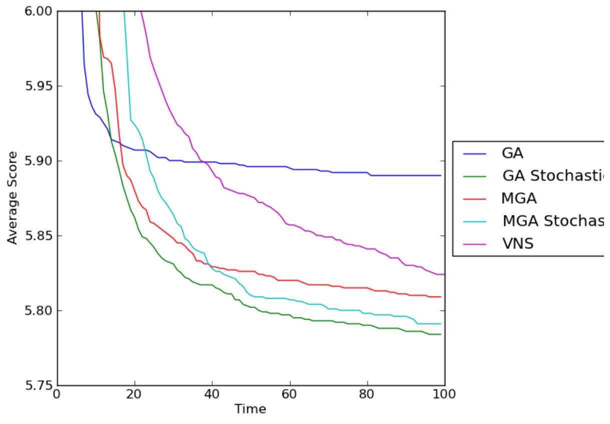
Parameters			Non-GA Heuristics		Original GA + variants						MGA + variants	
n	l	d	MVCA	VNS	Original	+SC	+SM	+KE	+FM	+All	Original	+All
25			0.003	0.036	0.171	0.176	0.170	0.157	0.172	0.172	0.384	0.388
50			0.009	0.173	0.256	0.271	0.256	0.242	0.257	0.266	0.681	0.575
100			0.011	0.798	0.400	0.414	0.396	0.377	0.396	0.412	1.325	1.175
250			0.042	5.73	0.673	0.693	0.682	0.647	0.665	0.722	2.939	2.756
500			0.116	24.21	0.992	1.001	1.029	0.962	0.981	1.073	5.354	5.283

(b) Running time(s)

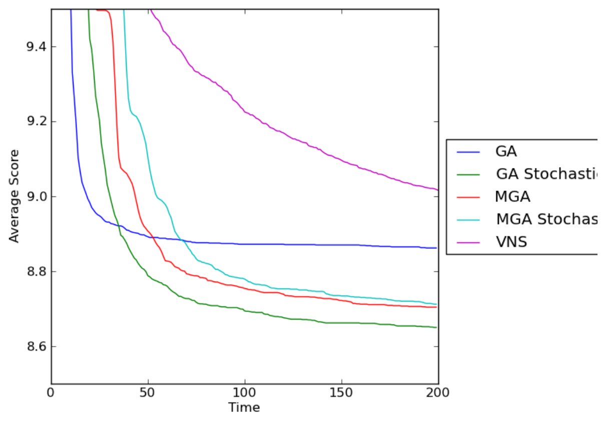
Table 1: Heuristic performance for various serial heuristics



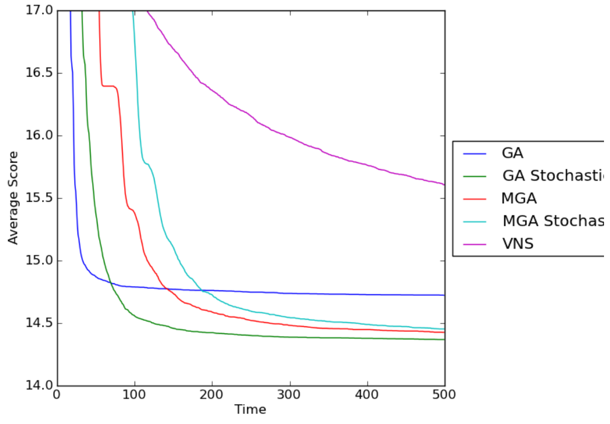
(a) L=25



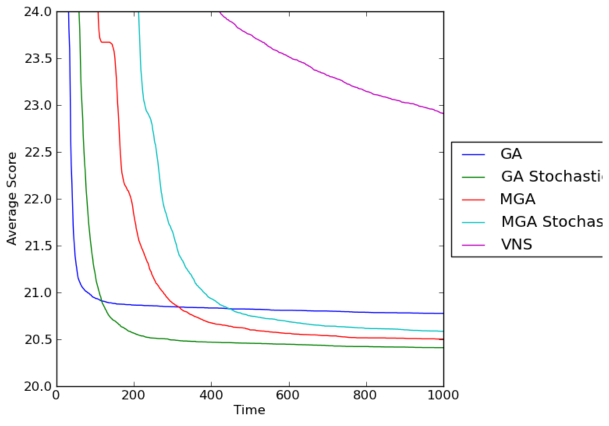
(b) L=50



(c) L=100



(d) L=250



(e) L=500

Figure 5: Average heuristic score vs Time for various serial GAs and the serial VNS. X axis is in cs (0.01s)

Instance Set	L=100	L=500
Time for serial (s)	2.38	5.42
Parallel-No Comm (s)	3.75	8.25
Parallel-Synchronized Iterations (s)	4.35	10.89
Parallel-Synchronized+Vault (s)	4.51	12.03

Table 2: Run times for Parallel Algorithms (Vault had vault size of 20 and vault breeding parameter 0.2)

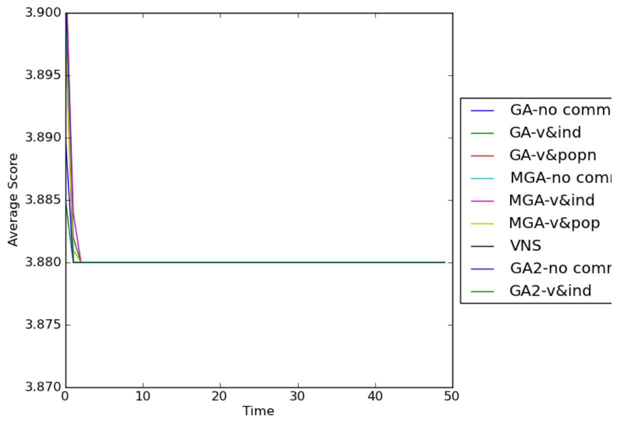
Class	Scheme	Average	Standard Deviation	$p < 0.05$	$p < 0.005$
No Communication		14.462	0.031	N/A	N/A
Global	Vault alone	14.439	0.028	Y	N
	Vault + ind	14.390	0.031	Y	Y
	Vault + pop	14.445	0.033	Y	N
Local	Replacement	14.462	0.036	N	N
	Waiting Room	14.453	0.024	N	N

(a) 8 processors

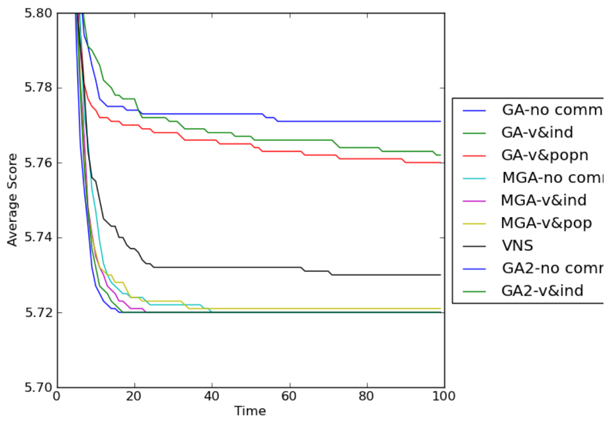
Class	Scheme	Average	Standard Deviation	$p < 0.05$	$p < 0.005$
No Communication		14.342	0.028	N/A	N/A
Global	Vault alone	14.309	0.033	Y	N
	Vault + ind	14.264	0.030	Y	Y
	Vault + pop	14.321	0.019	Y	N
Local	Replacement	14.331	0.023	N	N
	Waiting Room	14.321	0.023	Y	N

(b) 32 processors

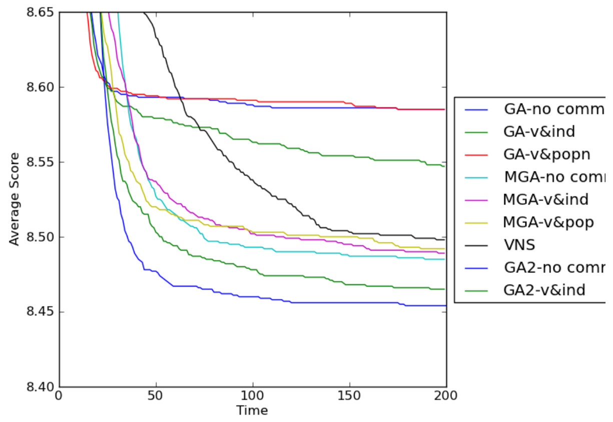
Table 3: Results for different Migration schemes (run over $L = 250$ set of instances for 20 generations with 10 repetitions. Last columns test for statistic significance over 'No Communication' using the Mann-Whitney U test)



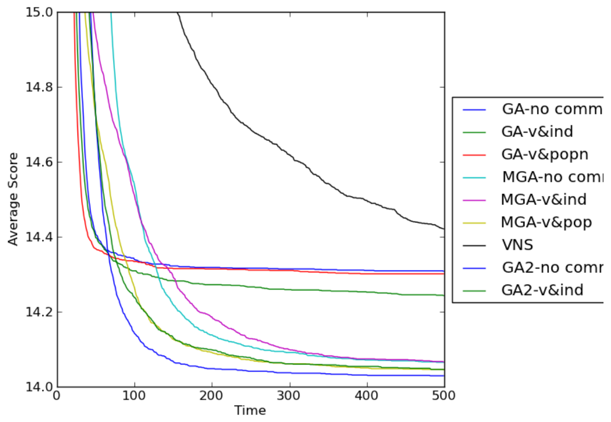
(a) L=25



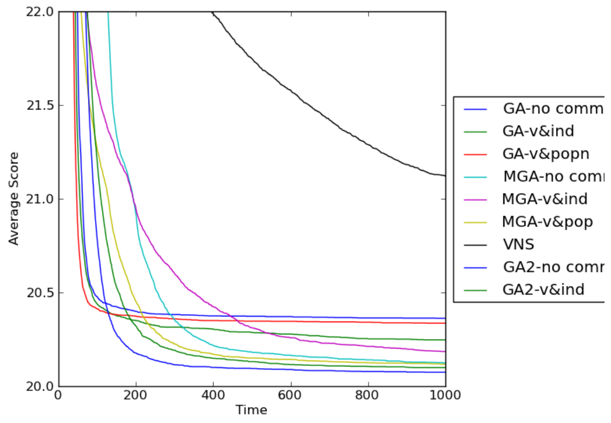
(b) L=50



(c) L=100



(d) L=250



(e) L=500

Figure 6: Average heuristic score vs Time for various parallel GAs and the parallel VNS, run on 32 processors. X axis is in cs (0.01s)