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

December 16, 2011

Abstract

Genetic Algorithms (GAs) have shown themselves to be very powerful tools for a variety of combinatorial optimization problems. Through this project I hope to implement a GA to solve the Minimum Labeling Spanning Tree (MLST) problem (a combinatorial optimization problem). Additionally, I intend to develop a parallel implementation of the GA, which will involve designing and testing various inter-processor communication schemes. The resulting parallel GA will be tested on a database of problems, where possible comparing results and running time with other serial heuristics proposed in the literature. Finally, the parallel heuristic will be analyzed to determine how performance scales with the number of processors.

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. Appendix B contains pseudo-code for one such heuristic, the Maximum Vertex Covering Algorithm (MVCA) by Chang and Leu [4]. Other heuristics that have been used include Simulated Annealing, Tabu Searches, Pilot Methods, Variable Neighborhood Searches and a Greedy Randomized Adaptive Search [3, 6]. I hope to be able to compare my algorithm with several of these heuristics.

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 perturbing 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 $(j + k)$ mod 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 3. 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 4. 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 which 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 $2l$ 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. This GA 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 selection, combination, mutation and replacement operators were modified

to cause and maintain a greater diversity in the population.

Several such techniques are included below. A discussion of their efficacy is included in the Results section.

4.1 Imposing a Population structure

One way to maintain diversity in the population is to limit the speed at which genetic material spreads through the population. One popular way to do this is to implement some structure on the population so that each individual can only breed with a subset of the entire population. This method to increase diversity in the population can be included in any genetic algorithm. Three different such population arrangements are discussed below (for more information see Alba&Dorronsoro [7]). Diagrams of each arrangement are included in Figure 2.

4.1.1 Panmictic

A Panmictic GA allows all individuals to breed with all other individuals. This allows genetic material to spread through the population very quickly, leading to reduced diversity in the population. Literature shows that this type of GA is the least likely to find the global minimum.

4.1.2 Distributed

Distributed GAs impose an island structure on the population - 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. In this case, more islands and a weaker migration operator (migration occurring more rarely) will lead to more diversity in the population. This population structuring has been shown to be generally more successful at finding the global minimum than a panmictic algorithm.

4.1.3 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 factor here is the size of the neighborhood compared with the size of the total population, with a smaller neighborhood impeding the spread of information and allowing more diversity in the population. These algorithms have been shown to be generally much more successful than panmictic algorithms and slightly better than distributed algorithms at finding global minima.

4.2 Further Modifications

A closer inspection on the design of Xiong's genetic algorithm led me to consider three further modifications. These methods are unique to the genetic algorithm outlined above in section 3, although similar modifications may be made to other genetic algorithms.

4.2.1 Coin toss

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. Currently these operators always favor 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 perturbing 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.

4.2.2 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 is hoped that this tactic will lead to less chance of the population getting stuck in a local minimum.

4.2.3 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 overcome this the mutation operator was modified such that the new gene is marked and I do not try 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 is 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 these reasons I shall use shared memory via Pthreads to parallelize my code.

6 Parallel GA code for the MLST

Next I considered a parallel implementation of a distributed genetic algorithm - a distributed genetic algorithm was chosen both because it is expected to outperform the existing panmictic algorithm and because it is easy to parallelize. I therefore start by breaking up the population into n subpopulations of equal size (assuming the total population size, p, is a multiple of n) and construct parallel code such that each subpopulation is allocated to its own thread (which are then distributed over all available processors). I consider three different migration operators which dictate how the various threads (subpopulations) will communicate with each other.

6.1 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 it should not have any effect on the genetic algorithms efficacy (although I do expect a speedup as I am now spreading the computational work over several processors).

6.2 Scharrenbroich 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 ... . In Scharrenbroich's genetic algorithm migration occurs when a subpopulation has stagnated (after K generations with no improvement), involving copying each of its neighbors best solutions into its own subpopulation (replacing its worst solutions).

6.3 Alternative Migration scheme

The above migration scheme replaces the weakest individual in each population, reducing diversity especially for small subpopulations. To avoid this I formulated a third migration scheme which does not involve replacing individuals. 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.

7 Implementation, Hardware

All heuristics will be encoded in C++ with POSIX threads (Pthreads) used for inter-processor communication. To date the code has only been run on a personal computer (2.2GHz quad-core Intel Core i7) although I have gained access to the GENOME cluster at the University of Maryland and intend to move the code to this cluster next semester.

8 Databases

The original paper by Xiong et al. [16] outlined a method to randomly generate connected, Iabeled 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. A subset of these were later used by Consoli et al. [6]. In order to be able to effectively evaluate my heuristic I intend to focus mainly on these sets used in both papers, although I will also be able to construct other graphs with different properties using the original algorithm should the need arise.

9 Validation and Testing

9.1 Validation of Serial Code

In order to validate my code I have obtained copies 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 should be able to ensure that exactly the same results are returned from both sets of computer code.

9.2 Validation of Parallel Code

To ensure that the migration operators are correctly encoded to carry the genetic material from one subpopulation to the others I intend to modify the algorithm to include a flag gene to indicate ultimate superiority (no need to worry about connectedness of graph) which should spread among all subpopulations (and all individuals) as time progresses.

9.3 Testing the Serial Implementation

The serial heuristics will then be run over the test instances (see 'Databases' above) to test performance both in terms of results and computational time. When run over the graphs from Cerulli et al. I will be able to compare my heuristics against each other, against the heuristics from Cerulli et al. [3] and Consoli et al. [6] as well as against the global optimum as found in Consoli et al. via an exhaustive search (only for some instances). If I do decide to run the heuristics over graphs that I might generate I will be able to compare my heuristics against each other and against a global optimum if it can be found in a reasonable amount of time by a global search.

9.4 Testing the Parallel Implementation

The parallel algorithm will be tested in two ways. The first is to compare the parallel implementation without communication with a serial implementation of the parallel code (with the one processor playing the role of all the various processors). If encoded correctly the parallel and serial versions should return the same results. This will then allow us to calculate the speedup due to parallelization (how much faster the parallel code runs compared with the serial code), which should increase in proportion to the number of processors involved.

The second way to test the parallel code will be to compare the results from the parallel implementation with results from various serial heuristics. Both will be run for the same amount of time on the same number of processors (with the serial heuristics being run independently on each processor with different random seeds). By allowing the processors to communicate with each other, it is hoped that my parallel implementation will be able to return better results than any of the serial heuristics.

10 Results and Discussion

10.1 Serial GA

Validation was carried out as described above. Initially both sets of code were run on exactly the same instance, verifying after every generation that the population was the same. This was found to be the case. As a second test of the validity of the code both were also run on 200 diverse instances. In all cases both codes returned the same set of labels (from the best individual in the population).

Testing Using a set of 36 instances from Cerulli et al. [3] I tested both Xiong's GA and MGA both in their original form and with various combinations of the improvements outlined above in Section 4. Appendix C Tables 2-4 contain a complete record of all the results and computational times for the various genetic algorithms, as well as the results from other heuristics and the global solution (when known) taken from the relevant papers [3, 6]. A summary of the performance of the various GAs is included in Table 1.

Three out of the four changes proposed in Section 4 led to a significant improvement in the computational results, decreasing the average number of labels required by about 2%. None of these changes led to a significant increase in running time as expected. Moreover all the changes can be used in the algorithm together, decreasing the average number of labels by a further 1%.

My new GA using the various modifications is found to perform at a similar level to the the Modified GA by Xiong [17], but without a significant increase in computational time (8% increase as compared with a 267% increase). By including all of my improvements (except the 'Coin Toss - Crossover') as well as Xiong's modified crossover operator from the MGA I can achieve an even stronger result, with a minor decrease in running time over Xiong's MGA.

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 GA

10.2.1 Without Migration

Validation No rigorous validation of the parallel GA has been carried out to date.

Testing Table 5 documents the results and speedup achieved by running the distributed genetic algorithm either on one processor or on four processors without any communication between subpopulations (no

migration between subpopulations). Comparing the two sets of results using the Mann-Whitney U-Test it was determined that in none of the cases was there a significant difference between the parallel and serial algorithms (with $\alpha = 0.05$). When we compare the results from dividing up the population we generally find that using more subpopulations (holding the total number of individuals constant) gives a weaker algorithm.

Finally by comparing the running times for the parallel and serial distributed algorithms we can see that significant speedup has been achieved, with an efficiency of over 80% in all cases (efficiency being the speedup divided by the number of processors, with a theoretical maximum of 100% according to Amdahl's Law). An interesting result here is that in some cases we are achieving an efficiency over 100% - this is due to the use of Hyper-Threading on i7 processors (allowing each processor to simultaneously process two threads). More information on Hyper-Threading can be found on Intel's website [10]. A plot showing how both the solution quality and computational time changes for differing numbers of subpopulations in the parallel implementation is included in Figure 6.

10.2.2 With Migration

Table 6 contains data gathered when the distributed algorithms were run again using the migration operators outlined above in Section 6. Once again the Mann-Whitney U-Test (with $\alpha = 0.05$) was used to test if the migration schemes led to a significant change in the performance of the heuristics. Only two cases, those with 8 and 16 islands using "Alternative" migration scheme, showed that the migration scheme caused a significant improvement.

There are several reasons why these operators are not giving the desired improvements. The most obvious flaw might be in the design of the migration schemes. I hope to be able to experiment with a great variety of schemes in the coming months to see if I can improve the heuristic in this way. A second problem might lie with the population sizes and the processing power currently used. For example Scharrenbroich [13] found his scheme worked well when run with 25 subpopulations each containing 50 individuals (1250 individuals in total) using 25 processors. In contrast I have been running my heuristic with at most 64 total individuals using 4 processors. This may be solved by moving my heuristic onto the GENOME cluster at the University of Maryland which has 32 processors, which I intend to do early in 2012. One final cause for the lack of success with migration operators might be the underlying heuristic running on each subpopulation (Xiong's GA). This might be causing all the individuals on the various processors to converge to the same or very similar solutions so that moving the best individuals between subpopulations does not in fact spread useful information. I may be able to remedy this by modifying other operators within Xiong's GA.

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 - Started**
  Tasks: Experiment with different population arrangements, migration operators to obtain optimal parallel GA. Possibly design and implement a Cellular GA for comparison
  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**
  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**: Time permitting, modify GA code and test on a different combinatorial optimization problem.

12 Deliverables

- Competitive, efficient 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.

- Final report and presentation analyzing technique and results.

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] Intel Corporation. Intel® Hyper-Threading Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, December 2011.

[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: Figures

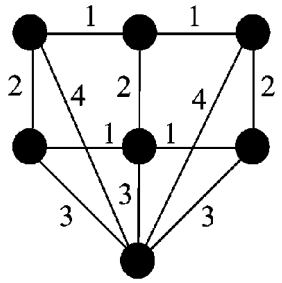

Figure 1: An example of a labeled spanning tree [from [16]]. In this case, the (unique) minimum set of colors which will generate a connected sub-graph is {2, 3}

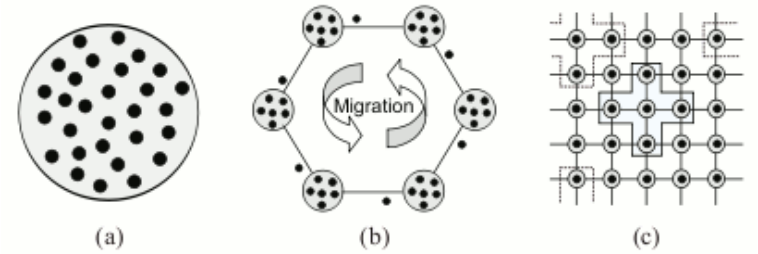

Figure 2: Three different population arrangements - a) panmictic, b) distributed, c) cellular genetic (CGA) [from [7]]. In each of these a dot represents an individual (solution) which can 'breed' with other solutions within its shaded neighborhood.

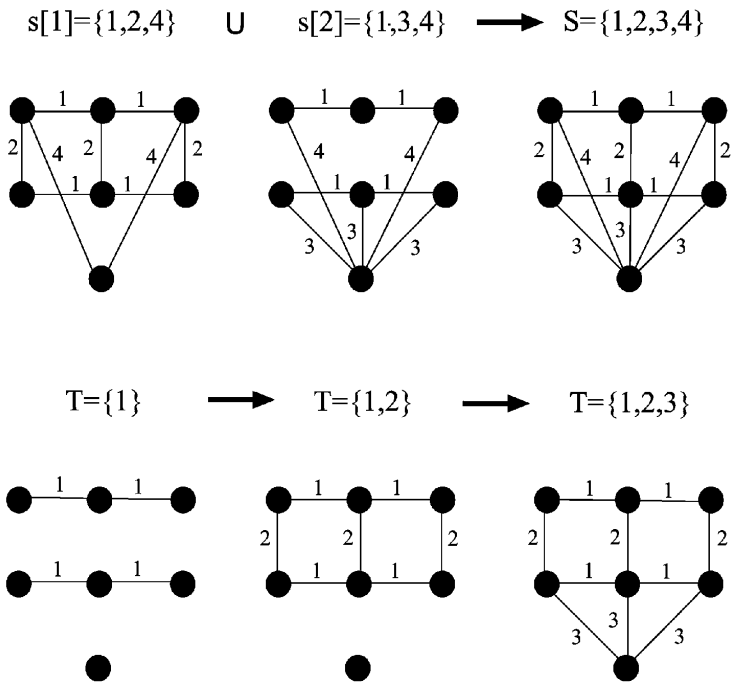

Figure 3: An example of Xiong's Crossover algorithm [16]

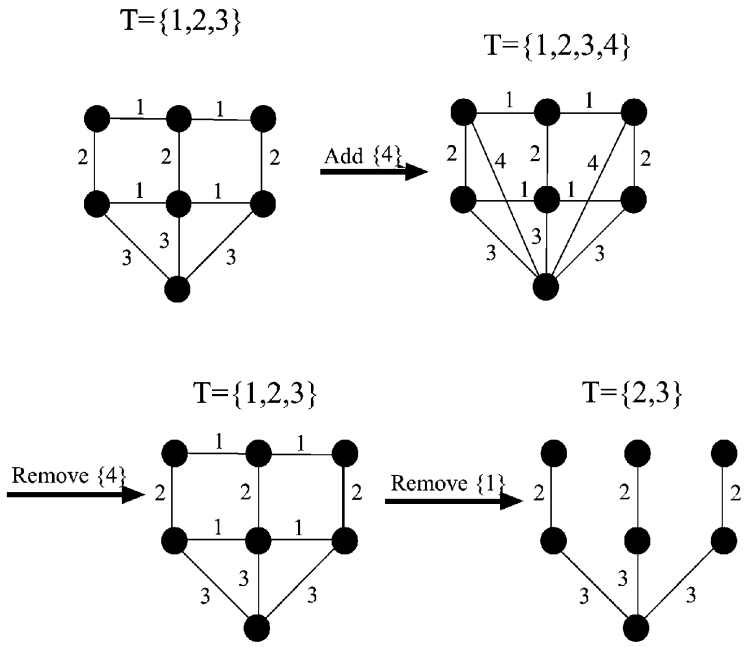

Figure 4: An example of Xiong's Mutation algorithm [16]

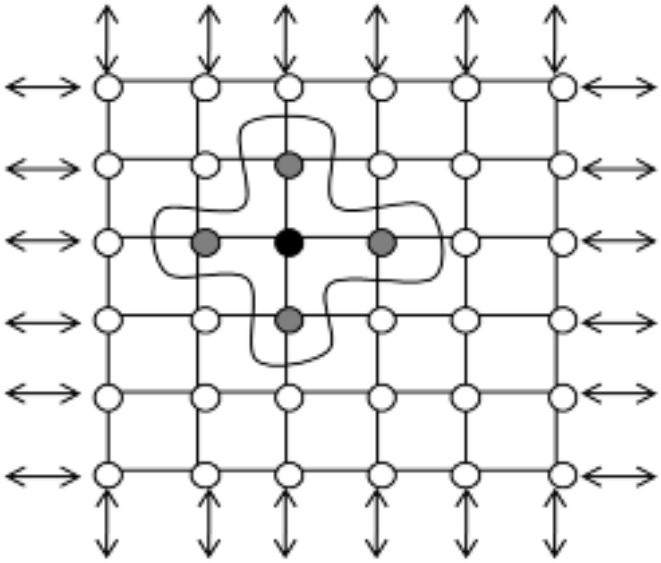

Figure 5: The Distributed Genetic Algorithm implemented by Scharrenbroich. Each 'node' now represents a processor, monitoring its own subpopulation, with the arrows representing migration between sub-populations [13]

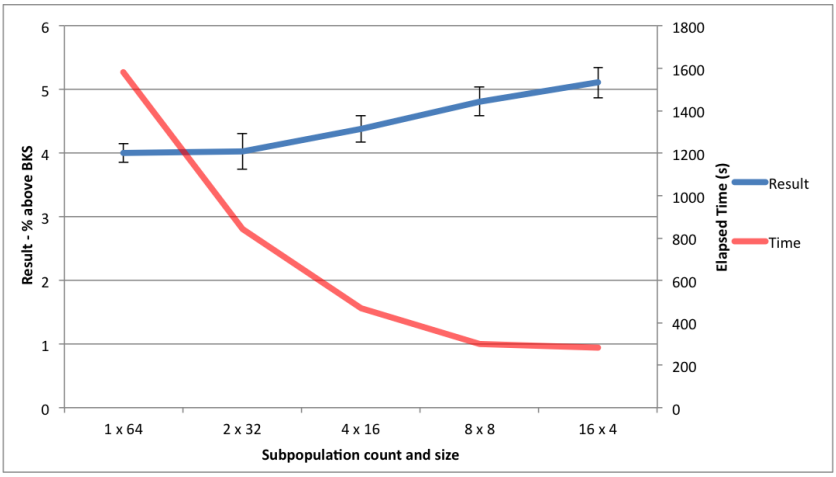

Figure 6: Solution quality and computational time for the Distributed Genetic Algorithm with varying numbers of subpopulations

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 $C = \emptyset$ be the set of used labels
Let H be the subgraph generated by C (H updated whenever C is)
While H not connected:
    For all colors $i \in L - C$:
        Add i to C
        Count the number of connected components in H
        Remove i from C
    Add the color which resulted in the least number of connected components

Crossover Operator from Xiong's GA

Let S be the union of all labels from either parent
$T = \emptyset$
Sort labels in Saccording to frequency in G While T is not viable:
    Add next color in S to T (first to last)

Mutation Operator from Xiong's GA

Let T be the set of labels belonging to the offspring (follows on from crossover operator)
Choose random $c \in L - T$
Add c to T
Sort labels in T according to frequency in G
For label in T (reverse iterate, start from least common in G):
    Remove label from T
    If T is viable (subgraph connected):
        continue
    Else:
        (Re-)Add label to T

Combination operator from Xiong's MGA

Let S be the union of all labels from either parent
Apply the MVCA algorithm (above) to the subgraph generated by S to get output

Appendix C: Results

Key to Tables:
GA- variants:
CT-C = Coin Toss-Crossover, CT-M = Coin Toss-Mutation, KE = Keep Equal, FM = Force Mutation, All = CT-C+CT-M+KE+FM.
Non-GA algorithms:
MVCA = Maximum Vertex Covering Algorithm from Chang and Leu, 1997 [4]
SA = Simulated Annealing, RTS = Reactive Tabu Search and Pilot = Pilot Method from Cerulli et al., 2006 [3] (NB Pilot Method results come from Consoli et al., [6]
GRASP = Greedy Randomized Adaptive Search Procedure and VNS = Variable Neighbourhood Search from Consoli et al. [6]

Comments on how results are recorded:
For each set of instances (10 instances in each set) we report the average number of labels returned by the heuristic as the result and the clock time to solve all 10 instances as the time. The results and times are then aggregated over all sets in a given group (Tables 2-4) or over all sets in all the groups (Tables 1, 5 & 6).

For Tables 5 and 6 (parallel algorithms) the algorithms were run 10 times with different random seeds and we report the average result, standard deviation in results and average time for these 10 trials.

In all serial cases the GAs were run with 20 individuals and with 20 generations (as done by Xiong [16]). For the parallel algorithms the population the populations and number of generations were both increased to 64 so that the population might be divided into many subpopulations of a reasonable size.

Table 1: Summary of Serial GA Results (best heuristic, VNS, is included for comparison)

Method	Total Result	Total Time(s)
Original GA	194.9	129.3
+CT-C	190.7	139.8
+CT-M	191.5	130.5
+KE	197.4	123.1
+FM	191.5	130.1
+All	188.5	139.7
Original MGA	188.7	474.8
+All	187.5	421.1
VNS	182.2	N/A

Table 2a: Computational results for Group .1

Parameters				Non-GA Heuristics						Original GA + variants						MGA + variants	
n	l	d	Optimal	MVCA	SA	RTS	Pilot	GRASP	VNS	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
20	20	0.2	2.4	2.6	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4
		0.5	3.1	3.5	3.1	3.1	3.2	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1
		0.8	6.7	7.1	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7
30	30	0.2	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
		0.5	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7
		0.8	7.4	8.0	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4	7.4
40	40	0.2	2.9	2.9	2.9	2.9	2.9	2.9	2.9	3.0	2.9	2.9	3.0	2.9	2.9	2.9	2.9
		0.5	3.7	3.9	3.9	4.0	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.8	3.7	3.7	3.7
		0.8	7.4	8.6	7.4	7.9	7.6	7.4	7.4	7.4	7.4	7.5	7.8	7.4	7.4	7.4	7.4
50	50	0.2	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0
		0.5	4.0	4.4	4.2	4.2	4.0	4.0	4.0	4.2	4.2	4.1	4.4	4.1	4.1	4.2	4.2
		0.8	8.6	9.2	8.7	8.8	8.6	8.6	8.6	8.7	8.6	8.6	8.8	8.6	8.6	8.6	8.6
Total			55.7	59.7	56.2	56.9	56.0	55.7	55.7	56.1	55.9	55.9	57	55.8	55.8	55.9	55.9

Table 2b: Time for Group 1

Parameters			Original GA + variants						MGA + variants	
n	l	d	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
20	20	0.2	0.11	0.10	0.10	0.11	0.11	0.11	0.16	0.15
		0.5	0.13	0.12	0.12	0.14	0.13	0.12	0.23	0.21
		0.8	0.21	0.21	0.22	0.21	0.21	0.22	0.47	0.46
30	30	0.2	0.17	0.17	0.16	0.16	0.17	0.17	0.29	0.26
		0.5	0.21	0.22	0.21	0.20	0.21	0.22	0.40	0.36
		0.8	0.36	0.36	0.36	0.35	0.36	0.35	0.93	0.89
40	40	0.2	0.26	0.26	0.27	0.23	0.26	0.25	0.45	0.42
		0.5	0.31	0.33	0.31	0.29	0.31	0.32	0.62	0.58
		0.8	0.56	0.56	0.57	0.54	0.56	0.55	1.57	1.48
50	50	0.2	0.36	0.37	0.36	0.33	0.37	0.37	0.68	0.57
		0.5	0.47	0.51	0.49	0.44	0.48	0.50	1.14	0.96
		0.8	0.93	0.95	0.88	0.86	0.91	0.94	2.82	2.69
Total			4.09	4.16	4.06	3.85	4.09	4.13	9.78	9.01

Table 3a: Computational results for Group 2

Parameters				Non-GA Heuristics						Original GA + variants						MGA + variants	
n	l	d	Optimal	MVCA	SA	RTS	Pilot	GRASP	VNS	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
100	25	0.2	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8
		0.5	2.0	2.1	2.0	2.0	2.0	2.0	2.0	2.1	2.0	2.0	2.0	2.0	2.0	2.0	2.0
		0.8	4.5	4.8	4.5	4.6	4.5	4.5	4.5	4.6	4.5	4.5	4.5	4.6	4.5	4.5	4.5
	50	0.2	2.0	2.1	2.0	2.1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
		0.5	3.0	3.2	3.1	3.2	3.0	3.0	3.0	3.2	3.0	3.2	3.2	3.1	3.1	3.2	3.1
		0.8	6.7	7.5	6.9	7.1	6.9	6.7	6.7	7.0	6.8	6.9	7.0	7.0	6.9	6.8	6.7
	100	0.2	3.0	3.3	4.0	3.4	3.0	3.0	3.0	3.5	3.3	3.3	3.6	3.1	3.2	3.3	3.2
		0.5	4.7	5.1	5.2	5.1	4.7	4.7	4.7	5.0	5.1	4.9	5.2	4.8	4.8	4.8	4.8
		0.8	NF	11.0	10.7	10.9	10.1	9.8	9.7	10.4	10.1	10.2	10.6	10.2	10.0	10.3	10.1
	125	0.2	4.0	4.0	4.1	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
		0.5	5.2	5.9	6.0	5.8	5.4	5.2	5.2	5.7	5.6	5.7	5.7	5.5	5.7	5.5	5.5
		0.8	NF	12.2	11.9	12.3	11.2	11.0	12.1	12.1	11.5	11.6	12.2	11.9	11.3	11.5	11.4
Total			NF	63.0	62.1	62.3	58.7	57.7	57.6	61.4	59.7	60.1	61.8	60.0	59.3	59.7	59.1

Table 3b: Time for Group 2

Parameters			Original GA + variants						MGA + variants	
n	l	d	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
100	25	0.2	0.64	0.64	0.65	0.64	0.65	0.69	1.10	1.11
		0.5	0.80	0.80	0.78	0.74	0.81	0.82	1.34	1.18
		0.8	1.67	1.62	1.64	1.60	1.67	1.66	3.87	3.24
	50	0.2	0.78	0.79	0.79	0.82	0.79	0.82	1.39	1.25
		0.5	1.09	1.25	1.15	1.17	1.15	1.28	2.40	1.93
		0.8	2.50	2.68	2.50	2.53	2.49	2.71	7.40	7.32
	100	0.2	1.36	1.37	1.39	1.26	1.30	1.40	2.82	2.43
		0.5	1.85	2.02	1.81	1.71	1.81	1.96	4.90	4.11
		0.8	3.62	3.73	3.50	3.54	3.46	3.79	15.38	13.75
	125	0.2	1.46	1.58	1.42	1.37	1.51	1.59	3.59	3.04
		0.5	2.14	2.28	2.09	2.06	2.02	2.43	6.29	5.17
		0.8	4.10	4.43	4.11	4.01	4.32	4.76	18.66	16.48
Total			22.02	23.20	21.82	21.44	21.97	23.90	69.15	61.02

Table 4a: Computational results for Group 3

Parameters				Non-GA Heuristics				Original GA + variants						MGA + variants	
n	l	d	Optimal	MVCA	Pilot	GRASP	VNS	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
200	50	0.2	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
		0.5	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
		0.8	5.2	5.6	5.2	5.2	5.2	5.6	5.3	5.2	5.7	5.4	5.3	5.3	5.2
	100	0.2	2.6	2.9	2.6	2.6	2.6	2.8	2.8	2.9	3.0	2.9	2.8	2.8	2.7
		0.5	3.4	4.0	3.4	3.4	3.4	3.9	3.6	3.9	4.0	3.8	3.9	3.6	3.6
		0.8	NF	9.0	8.3	8.1	7.9	8.9	8.5	8.7	8.8	8.6	8.6	8.5	8.5
	200	0.2	4.0	4.0	4.0	4.0	4.0	4.1	4.0	4.0	4.1	4.1	4.0	4.0	4.0
		0.5	NF	6.4	5.5	5.4	5.4	6.3	6.4	6.1	6.4	6.4	6.0	6.2	5.8
		0.8	NF	13.4	12.4	12.2	12.0	13.9	13.2	13.3	14.2	13.3	13.0	12.9	12.7
	250	0.2	4.0	4.4	4.0	4.1	4.0	4.8	4.8	4.9	4.9	4.7	4.8	4.9	4.5
		0.5	NF	7.2	6.3	6.3	6.3	7.3	7.0	7.0	7.3	7.2	6.8	6.9	6.8
		0.8	NF	15.7	13.9	13.9	13.9	15.6	15.3	15.1	16.0	15.1	14.8	14.6	14.5
Total			NF	76.8	69.8	69.4	68.9	77.4	75.1	75.5	78.6	75.7	74.2	73.9	72.5

Table 4b: Time for Group 2

Parameters			Original GA + variants						MGA + variants	
n	l	d	Original	+CT-C	+CT-M	+KE	+FM	+All	Original	+All
200	50	0.2	2.48	2.69	2.63	2.45	2.56	2.49	4.62	4.48
		0.5	3.01	3.11	3.09	3.09	3.18	3.13	5.92	5.13
		0.8	7.64	8.37	7.78	7.19	7.47	7.47	19.95	17.56
	100	0.2	3.83	4.06	3.66	3.46	3.88	3.95	8.34	7.47
		0.5	5.00	5.60	5.02	4.67	5.44	5.34	11.38	11.65
		0.8	11.42	12.93	11.92	11.06	11.41	13.16	45.84	39.50
	200	0.2	5.46	6.12	5.35	5.22	5.54	5.84	13.61	11.85
		0.5	8.06	9.00	8.77	7.76	8.55	8.87	28.46	21.89
		0.8	18.97	19.72	18.67	17.10	18.48	19.96	89.70	84.52
	250	0.2	6.77	7.05	6.71	5.93	6.63	7.34	18.20	14.98
		0.5	9.54	11.39	9.75	9.33	9.90	10.35	34.51	29.87
		0.8	20.96	22.42	21.30	20.54	21.02	23.77	115.37	102.17
Total			103.14	112.47	104.63	97.78	104.06	111.66	395.91	351.07

Table 5: Serial and Parallel Distributed GA (no migration)

Subpopulation Size & Count	Serial Algorithm			Parallel Algorithm			Speedup	Efficiency
	Mean	Std dev.	Time	Mean	Std dev.	Time		
64 x 1	189.49	0.27	1583	N/A	N/A	N/A	N/A	N/A
32 x 2	190.08	0.39	1555	189.63	0.54	843	1.84	92%
16 x 4	190.39	0.25	1536	190.18	0.37	468	3.28	82%
8 x 8	190.91	0.61	1539	190.97	0.42	299	5.15	129%
4 x 16	191.39	0.33	1506	191.51	0.44	283	5.32	133%

Table 6: Different Migration schemes

Subpopulation Size & Count	No migration		Scharrenbroich			Alternative		
	Mean	Std dev.	Mean	Std dev.	Significant?	Mean	Std dev.	Significant?
32 x 2	189.54	0.51	189.80	0.33	N	189.70	0.31	N
16 x 4	190.18	0.37	190.12	0.33	N	189.91	0.46	N
8 x 8	190.97	0.42	190.57	0.46	N	190.23	0.49	Y
4 x 16	191.51	0.44	191.67	0.48	N	190.25	0.20	Y