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

October 11, 2011

Abstract

Genetic Algorithms (GAs) have shown themselves to be very powerful tools for a wide 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). If time permits, I may attempt to modify the code to solve another combinatorial optimization problem. Additionally, I will develop a parallel implementation of the GA, which will involve designing and testing various inter-processor communication schemes. The eventuating parallel GA will be tested on a database of problems, 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. We denote $|E| = e$, $|V| = v$ and $|L| = l$. One such graph is shown in Figure 1. A sub-graph is generated by only using 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 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)

1 - Initialization: The first step in a genetic algorithm is to create the original generation of feasible (valid) and varied solutions. At this stage we are not concerned with their 'fitness'.
2 - Selection: The next step is to choose pairs of individuals ('parents') from the current population which are to be bred. Simpler strategies include iterating through all possible combinations and randomly choosing sets of parents. Goldberg [10] 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).
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 get stuck away from the global minimum.
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].
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.
6 - Termination: A termination condition determines when we leave the 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).

2.2 Population Arrangement

Maintaining diversity is crucial in the success of a GA. This can be done by limiting 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. Three different such populations arrangement are discussed below (for more information see Alba&Dorronsoro [7]). Diagrams of each arrangement are included in Figure 2.
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.
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 form 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 much more successful at finding the global minimum than a panmictic algorithm.
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.

3 An existing Genetic Algorithm for the MLST

In 2005, Xiong et al. [16] implemented a genetic algorithm 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, with only one parameter and no fine-tuning. We 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 - Initialisation: 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 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 offsprings set of labels, and 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 we 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 was 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.

4 Extensions to the Serial GA code

The first part of my project will be to work on developing my own serial GA, using Xiong's GA as a starting point. This will be done by investigating how GAs have been used in the wider literature to solve combinatorial optimization problems, considering how the various selection, combination and mutation operators have been designed to achieve the best results. Where appropriate I will implement and test the alternative operators (or any of my own design), keeping those which are most suited to and perform best on the MLST problem. I will also experiment with population arrangements (discussed above) to achieve the best results.

The purpose for this is two-fold. One reason is obvious; I hope to be able to design competitive code which achieves strong results without a significant increase in computational effort (as compared with Xiong's GA). The second reason is more subtle; by playing with the various operators involved, I hope to be able to achieve a better understanding of how a variety genetic algorithms work, enabling me to better encode the more complicated parallel implementation.

5 Designing Parallel code

The second step will be to develop a parallel implementation of the genetic algorithm. The main reason for such a parallel algorithm is speed; by using a large number of processors running at or near capacity we expect better results in less 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 in turn 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 much more difficult to design (all processors must always be ready to reply to any requests from other processors while simultaneously running their own algorithm and deciding when to send out their own requests), but should operate at higher speeds.

6 Parallel GA code for the MLST

I hope to be able to implement both synchronous and asynchronous version of my GA, using direct communication between processors. The main task in this part of the project, after getting the inter-processor communication working, will be to investigate different communication strategies between the processors.

One option was put forward by Scharrenbroich [13]. This is essentially a distributed (island) model in which each island is allocated to its own processor. A mesh arrangement was used to arrange the processors such that migration only happens between neighboring processors (see Figure 5). By modifying migration operators (what migrates and how often), termination conditions and the mesh arrangements I hope to be able to achieve very strong results. A quick search of the literature reveals several other strategies [7–9] which might also prove useful in optimizing my code.

7 Implementation and Hardware

All heuristics will be encoded in C++, and for the parallel implementations I will use the Message Passing Interface (MPI). Initially the code will be run on a personal computer, although I hope to be able to gain access to the Genome cluster at the University of Maryland to try the code on a large array of processors.

8 Validation and Testing

8.1 Databases

Several sets of connected, labelled graphs will be generated for the purposes of testing and comparing the various heuristic. Each set will have a unique combination of parameters (v, e, l) and will containing many (≈ 100) different graphs with the same parameters. Modifying these parameters will allow us to control how easy or hard each set of graphs is to solve.

8.2 Testing the Serial Implementation

My serial heuristic will then be run over the sets to test performance both in terms of quality of results and computational time. It will be compared both with heuristics from the literature and with the true global minimum (if it can be found through an exhaustive search in a reasonable amount of time).

8.3 Testing the Parallel Implementation

The parallel algorithm will be tested in two ways. The first is by comparing it 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 exactly the same results. This will then allow us to calculate the speed-up due to parallelization (how much faster the parallel code runs compared with the serial code), which should increase proportional 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.

9 Schedule

- **Create my serial GA**
  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**
  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 speed-up against serial code

- **Fine tuning parallel GA**
  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, 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.

10 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] R. Cerulli, A. Fink, and M. Gentili. 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] Sven E Eklund. A massively parallel architecture for distributed genetic algorithms. *Parallel Computing*, 30:647–676, 2004.

[10] 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.

[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. Available at UMBC Library.

[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

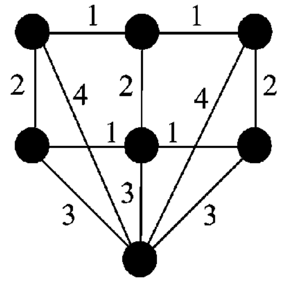

Figure 1: An example of a labeled spanning tree [from [16]]. In this case, the (unique) minimum set of colors which will generate a connected sub-graph is {2, 3}

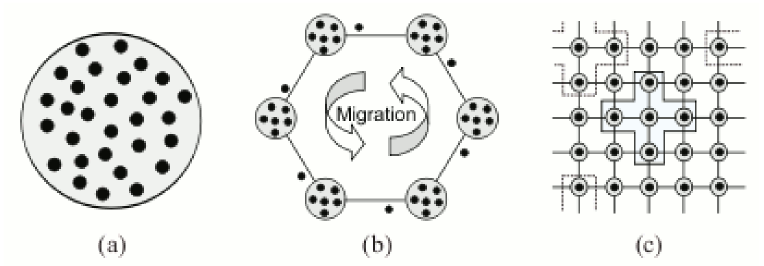

Figure 2: Three different population arrangements - a) panmictic, b) distributed, c) cellular genetic (CGA) [from [7]]. In each of these a dot represents an individual (solution) which can 'breed' with other solutions within its shaded neighborhood.

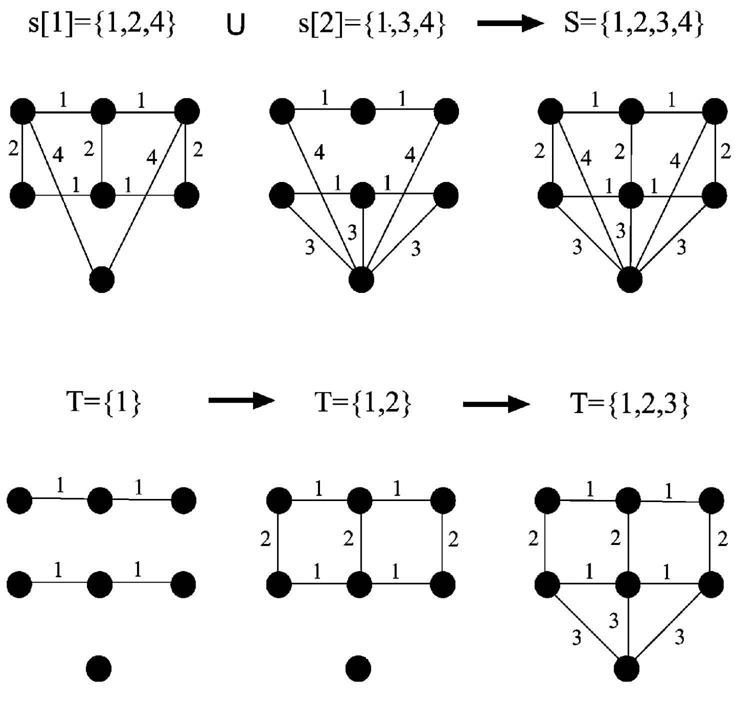

Figure 3: An example of Xiong's Crossover algorithm [16]

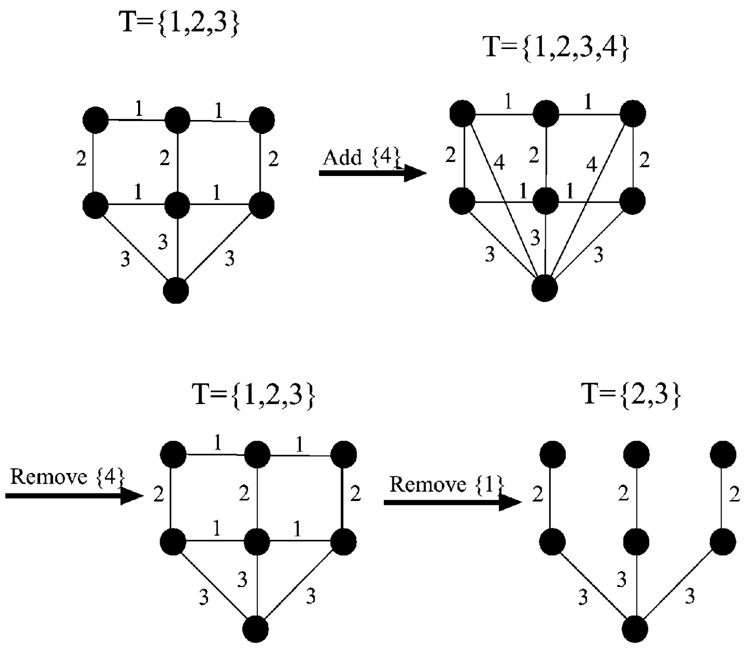

Figure 4: An example of Xiong's Mutation algorithm [16]

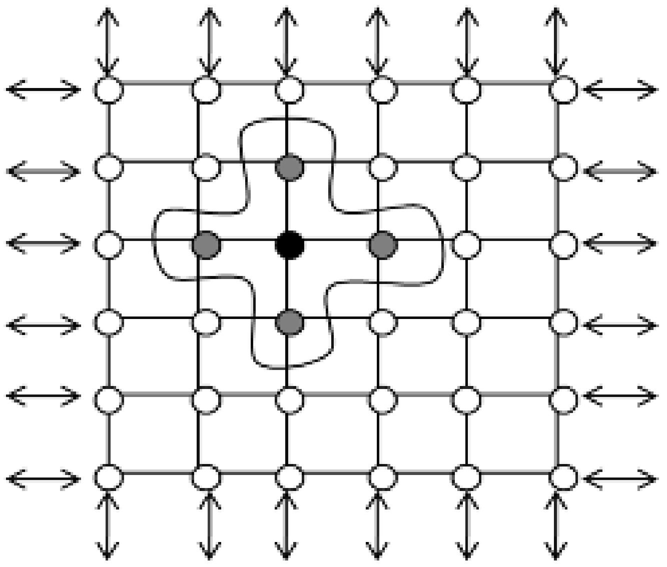
Figure 5: The Distributed algorithm implemented by Scharrenbroich. Each 'node' now represents a processor, monitoring its own sub-population, with the arrows representing migration between sub-populations [13]

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 $C = \emptyset$ 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 $S$according 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
```