GRAPH PARTITIONING USING LEARNING AUTOMATA

B. John Oommen[+] **and Edward V. de St. Croix**
School of Computer Science
Carleton University
Ottawa ; Canada : K1S 5B6.

ABSTRACT

Given a graph **G**, we intend to partition its nodes into two sets of equal size so as to minimize the sum of the cost of the edges having end-points in different sets. This problem, called the uniform Graph Partitioning Problem (GPP), is known to be NP-Complete. In this paper we propose the first reported learning-automaton based solution to the problem. We compare this new solution to various reported schemes such as the Kernighan-Lin's algorithm, and two excellent recent heuristic methods proposed by Rolland *et. al.* -- an extended local search algorithm and a genetic algorithm. The current automaton-based algorithm outperforms all the other schemes. We believe that it is the fastest algorithm reported to date. Additionally, our solution can also be adapted for the GPP[1] in which the edge costs are not *constant* but random variables whose distributions are *unknown*.

Keywords : Heuristic search; Graph partitioning; Adaptive learning; Learning Automata.

I. INTRODUCTION

Let **G** = (V, E) be a graph where the number of nodes, V, is even. We intend to partition V into two disjoint node sets V_1 and V_2 in such a way that the sum of the cost of edges having end-points in different sets is minimized. This problem, referred to as the Graph Partitioning Problem (GPP), has been known to be NP-Complete [GJ79,GJS76]. The solution to the GPP can be used in several important areas. Electronic circuit board communication rates are optimized when the distances between the components are minimized. Minimal distances between components are also required in order not to skew the communication timing delays. On a larger scale, wide area networks may show better performance if nodes are clustered around the database-servers which they access the most. This same principle applies to computer memory paging where the ideal situation is to minimize disk swapping by clustering inter-related subroutines and data on the same memory pages. Similarly, the problem of laying out complex floor space in industrial settings, and allocation problems in distributed and parallel computing environments can be seen to be directly related to the GPP.

Due to the inherent complexity of solving the GPP a whole body of literature reporting various heuristic solutions has emerged. Rather than survey the entire field, but in the interest of brevity, we refer the reader to an excellent thesis [Ro91] and some recent papers [PR94,RP92] which give an up-

---

[+] Senior Member, IEEE. The work of this author was partially supported by the Natural Sciences and Engineering Research Council of Canada.

[1] With regard to this claim, please see the footnote at the end of Section I (on Page 3).

to-date survey of the results available and of the various algorithms that have been currently reported. One of the pioneering works in this direction was the solution due to Kernighan and Lin [KL70]. Their scheme, Algorithm KL, has been universally considered to be among the best solutions. More recently [JAM89], Johnson *et. al.* showed how the philosophy of simulated annealing [AK89] (referred to Algorithm SA) could be adapted to produce excellent solutions for the GPP. Their paper demonstrated how the solution generated by Algorithm KL could be significantly improved. Unfortunately, though, Algorithm SA is more computationally burdensome than Algorithm KL and was reported [FK90] to be inferior to other methods when the GPP involved partitioning V into K subsets (the K-way GPP). The results of [FK90] also show the extensive computational needs of Algorithm SA and its relatively poor performance in various settings.

To our knowledge, the best reported algorithms to date, those proposed by Rolland[2] *et. al.* [PR94,Ro91,RP92] are :

(i) An extended local search scheme (referred to as Algorithm XLS), and

(ii) A Genetic Algorithm (referred to as Algorithm GA).

The above references demonstrate the power of these two strategies.

In this paper we shall present a new technique for solving the GPP. This involves the use of stochastic learning automata. Learning Automata (LA) have been used to model biological learning systems and also to learn the optimal action which a random environment offers. Learning is achieved by interacting with the environment and processing its responses to the actions that are chosen. Such automata have various applications such as parameter optimization, statistical decision making and telephone routing [NT89,OF93,OM88,OVZ92,OZ93]. An excellent book by Narendra and Thathachar [NT89] contains a review of the families and applications of learning automata.

The learning process of an automaton can be described as follows: The automaton is offered a set of actions by the environment with which it interacts, and it is constrained to choose one of these actions. When an action is chosen, the automaton is either rewarded or penalized by the environment with a certain probability. A learning automaton is one which learns the optimal action, which is the action that has the minimum penalty probability. Hopefully, the automaton will eventually choose this action more frequently than other actions.

By suitably modelling the graph weights and feasible solutions as an "Environment" we shall use the principles of LA to develop a very efficient solution for the GPP. Indeed, our current automaton-based algorithm outperforms all the other schemes catalogued above by an order of magnitude. We believe that it is the fastest algorithm reported to date. Finally, our solution is the first reported

[2]We are very grateful to Professor Rolland for introducing us to this problem. We appreciate his quick e-mail responses to our queries. Our comparative results could *never* have been obtained if it had not been for his unusually good and cooperative attitude of mind. Indeed, in that sense, he was more of a co-worker in the publication of this paper than an observer.

solution which can be adapted for the GPP when the edge costs are not *constant* but random variables with *unknown* distributions.[3]

The organization of the paper is as follows. Section II reviews the related work on the GPP and in particular, the principles and algorithms referred to above are *briefly* reviewed. Section III outlines the use of a particular LA, the Object Migrating Automaton, which is the foundation for our present solution. We then introduce our solution, the Graph Partitioning Learning Automaton (GPLA) and conclude the paper by presenting various results demonstrating the power of our scheme.

II. PROBLEM FORMULATION AND PREVIOUS SOLUTIONS

II.1 The Graph Partitioning Problem (GPP)

Let G = (V, E) be any graph with an even number (2N) of vertices, V. The GPP[4] involves partitioning V into two node sets V_1 and V_2 (i.e., $|V| = |V_1| + |V_2|$ and $V_1 \cap V_2 = \Phi$) such that the sum of the edge-cost having end-points in different sets is minimized. Indeed, if c_{ij} is the symmetric cost of the edge connecting nodes i and j, the GPP is the following nonlinear optimization problem :

$$\text{Minimize} \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_i \cdot (1 - x_j) \tag{1}$$

$$\text{Subject to} \sum_{i \in V} x_i = N \tag{2}$$

$$\text{where } x_i \in \{0, 1\} \text{ for all } i \in V \text{ ; and } x_i = 1 \Rightarrow i \text{ is in set } V_1 \text{ ; } x_i = 0 \Rightarrow i \text{ is in set } V_2. \tag{3}$$

The formulation of the GPP can be alternatively rewritten in an unconstrained form as:

$$\text{Minimize} \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_i \cdot (1 - x_j) + \Pi \tag{4}$$

where Π is a penalty measure associated with (2). The latter formulation was explicitly utilized by Johnson *et. al.* in [JAM89] and later in the genetic algorithm proposed by [Ro91,RP92].

In the absence of a systematic algorithm for tackling the problem (which is known to be NP-Complete [GJ79,GJS76]), the most obvious strategy is to resort to a brute force exhaustive search of the solution space. In this case, the solution space is prohibitively large for any meaningful problem; furthermore, this space grows exponentially with the number of nodes. Indeed, when |V| = 10, the solution space is of cardinality 126. When |V| = 100 the space has more than 10^{29} feasible solutions !! Rather than attempt to obtain the optimal solution, most "approximation algorithms" try to produce

---

[3]The claim that our solution can be adapted for the GPP when the edge costs are not *constant* but random variables with *unknown* distributions has not been substantiated experimentally here. But, since all the learning principles invoked [OM88, NT89,Ts73] here are originally derived from the setting of interacting with *stochastic* environments, we believe that our claim will hold. We are still investigating how this claim can be verified and proven. The difficulty lies in a very fine technical issue : Unlike what is currently done in the literature concerning the GPP, experimentation with stochastic environments would involve taking the *ensemble* average of the time averages as it done in adaptive learning [NT89].

[4]The K-way GPP involves partitioning V into K disjoint subsets such that the sum of the cost of edges having end-points in different sets is minimized.

near-optimal solutions. Indeed, a whole body of literature has gone into designing heuristic strategies that yield solutions that are "arbitrarily close" to the optimal one, and which are computable in a "reasonable" amount of time. In order to present our results in the context of the best known solutions, we shall in the next few subsections, *briefly* catalogue some of these.

II.2 The Kernighan-Lin Algorithm

For many years the solution proposed by Kernighan and Lin [KL70] (referred to as Algorithm KL) was reckoned to be among the best heuristic algorithms. The salient feature of Algorithm KL is its ability to quickly locate "good" solutions. It uses the idea that some vertices are more strongly connected than others implying that these, typically, have a higher degree (which is certainly true for sparse graphs) and therefore should be migrated between potential solution sets as the clusters develop. Thus, this procedure essentially makes use of the observation that inter-connected nodes with weighty edge costs tend to cluster. Consequently, although migrating individual nodes between the sets may have a detrimental effect to the overall solution, Kernighan and Lin observed that by swapping groups of nodes between the intermediate solution sets, we can, hopefully, obtain solutions superior to those which are obtained by swapping individual pairs of nodes. In every iteration, the algorithm essentially first builds a set of gains (i.e., cost minimizations) obtained by swapping individual pairs of nodes and subsequently swaps all the node pairs found to contribute to this gain to generate a new solution[5]. This process is continued until no positive gains are obtained.

A precise description of the algorithm is given in [KL70,RP92]. In the implementation for which we have reported experimental results, at each stage the algorithm can either iterate on a new random solution or it can alternatively perform a sub-partitioning on the two solution partitions V_1 and V_2 and swap them to create V_1' and V_2'. This new solution then forms the basis for the next iteration. Typically, a counter (on the number of iterations executed) is used to terminate the scheme.

Although the algorithm is fast and reasonably accurate Pirkul and Rolland [Ro91,RP92] report that the solutions obtained by Algorithm KL are not consistently of good quality.

II.3 The Extended Local Search Algorithm

We shall now briefly describe the Extended Local Search (XLS) Strategy introduced by Rolland *et. al*. The latter is closely related to a local search scheme (Algorithm LS) which operates on a current partitioning and modifies it by migrating a pair of nodes between the individual sub-partitions. To be more specific, suppose that the current partition at time 'n' is $\pi(n)$. A solution is defined to be in the *neighborhood* of π if it can be obtained by swapping pairs of nodes between the

[5]Philosophically, the algorithm has two looping constructs. The outer loop terminates the scheme when no further improvement is obtained. The inner loop, on the other hand, constructs the set of nodes to be swapped. The interesting features of the technique are the details of how this loop migrates the nodes and how it permits the movement of nodes which increase the weighted cut-set sum. We refer the reader to the excellent "pioneering" work in [KL70] for these details. Although these have been implemented in our study, to avoid repetition the details are omitted in the interest of brevity.

two sub-partitions with the additional restriction that each node can be moved exactly once. Thus Algorithm LS searches the neighbourhood of $\pi(n)$ to find a local optimum, and so, given any feasible solution, a superior solution in its neighbourhood can be obtained by swapping a *single* pair of nodes. This is exactly the philosophy motivating Algorithm LS.

The number of swaps done per iteration is bounded by N since a node, once swapped, is no longer eligible for swapping. By re-initializing the scheme from an ensemble of random starting solutions large sections of the problem space can be searched. LS has the following properties :

(i) Due to the exponentially large size of the solution space, repeated invocations to Algorithm LS may not succeed in identifying an optimal solution [KL70].

(ii) Repeated invocations of Algorithm LS from an ensemble of random initial partitions has been shown to be more efficient than a simulated annealing approach for the traveling salesman problem [GS86].

(ii) Due to the existence of numerous "almost optimal" solutions for dense graphs, repeated invocations of Algorithm LS works well in the case of such graphs [Ro91].

Rolland *et. al*. enhanced Algorithm LS (in a scheme referred to as Algorithm Extended Local Search (XLS)) by permitting multiple swaps per iteration. Quite simply stated, the generalization consists of the following: Once a new solution has been obtained, the set of pairs to be swapped, P, is reconstructed repeatedly for every current best solution, and a node, once swapped, is permitted to be swapped again. Of course, looping is prevented by never moving to a more expensive solution.

Experimental evidence [Ro91,PR94,RP92] shows that the algorithm yields good quality solutions in a reasonable time. Since the number of swaps is not bounded by the number of elements in each sub-partition, Algorithm XLS is typically more time consuming than Algorithm LS. Also, since the number of iterations done by the former is unpredictable, in the worst case, unless a stopping criteria is explicitly imposed, all possible potential solutions could be investigated. A faster version of Algorithm XLS can be devised by creating the set of pairs to be swapped, P, for each iteration, but utilizing only the best node-pair to update the partitions. Note that, in this case, the procedure to construct P would not require sorting. Since the details these techniques are well documented in [Ro91,PR94,RP92] they are not re-iterated here in the interest of brevity. For the sake of experimental comparison, however, Algorithm XLS has been implemented and rigorously tested in our current study.

II.4 Genetic Algorithms for the GPP

A new philosophy to solve a variety of optimization problems emerged with the introduction of Genetic Algorithms (GAs). These algorithms (analogous in spirit to the more traditional "generate and test" algorithms) are motivated by the principles of the mechanics of natural selection. As opposed to a virgin "generate and test" algorithm which repeatedly generates potential solutions and

subsequently evaluates them, GAs generate potential solutions derived from solutions which were previously well ranked and mutated by "genetic" operations such as substitution, bit swapping and crossover. GAs are robust and effective for solving NP-complete problems [DS89], but their effectiveness is greatly dependent on the "GA Representation" of the solution space. They are inherently parallelizable since numerous solutions can be generated and tested simultaneously.

The literature in the field of GAs is extensive. We refer the reader to [CDQ91,Go87,Go89, LH89] for good reviews of the theory and relevant applications of GAs.

Pirkul and Rolland [Ro91,RP92] devised an Extended Genetic Algorithm (Algorithm XGA) to solve the GPP. The motivation for their *representation* can be found in [RP92]. Algorithm XGA used a population of N_{org} organisms with N_{gen} generations where each organism was a binary array of bits of size 2N. The individual bits represented the membership of the nodes as follows:

$x_i = 1 \Rightarrow$ i is in set V_1; $x_i = 0 \Rightarrow$ i is in set V_2.

In each generation the fittest organism generated (the best potential solution) was "coronated" to be the Queen, and a King was selected from the population using a weighted random selection process. The Queen and King were used to generate a pair of offspring using bit swapping and the two new offspring generated were subsequently mutated in a bit-wise manner so as to provide a more diverse and stronger gene pool for the next generation. Consequently, each subsequent generation could be expected to contain a larger proportion of the local optimal solutions. On termination of the algorithm (after a number of generations) the Queen represented the best solution to the GPP. Note that the description of GAs described here differs marginally from the more standard form in which the mating *pairs* (and not just the "kings") are selected randomly using the fitness scores to weight their selection probability. The scheme described for the GPP is formally given in [Ro91,RP92].

We shall now proceed to discuss our automaton-based solution to the GPP.

III. LEARNING AUTOMATA AND OBJECT PARTITIONING

Stochastic learning automata can be classified into two main families : (a) Fixed structure stochastic automata and (b) automata whose structures evolve with time. Examples of the former type are the Tsetlin, Krinsky and Krylov automata [NT89,Ts73,OM88]. Although the latter automata are called variable structure stochastic automata because their transition and output matrices are time varying, in practice, they are merely defined in terms of action probability updating rules [NT89].

A Fixed Structure Stochastic Automaton (FSSA) is a quintuple $(\alpha, \Phi, \beta, F, G)$ where :

(i) $\alpha = \{\alpha_1, ..., \alpha_R\}$ is the set of actions that it must choose from.

(ii) $\Phi = \{\phi_1, ..., \phi_S\}$ is its set of states.

(iii) $\beta = \{0, 1\}$ is its set of inputs where '1' represents a penalty and '0' a reward.

(iv) $F : \Phi \times \beta \rightarrow \Phi$ is a map called the transition map. It defines the transition of the state of the automaton on receiving an input. F may be stochastic.

(v) $G : \Phi \rightarrow \alpha$ is the output map and determines the action taken by the automaton if it is in state ϕ_i. With no loss of generality G is deterministic [NT89].

The selected action serves as the input to the environment which in turn emits (communicates) a stochastic response $\beta(n)$ at time 'n'. $\beta(n)$ is an element of $\beta = \{0,1\}$ and is the feedback response of the environment to the automaton. The environment penalizes (i.e., $\beta(n) = 1$) the automaton with the penalty c_i, which is action dependent. On the basis of the response $\beta(n)$, the state of the automaton $\phi(n)$ is updated and a new action chosen at (n+1). Note that the $\{c_i\}$ are unknown initially and it is desired that as a result of interaction with the environment the automaton arrives at the action which presents it with the minimum penalty response in an expected sense.

In this paper we propose that the GPP be solved by viewing the problem not as a searching problem or a parameter-based training problem, but instead as one that falls in the domain of object partitioning problems. The goal is to model the problem so as to determine the nodes in V that "match" other nodes possessing "similar" properties. In other words we attempt to *impose* an intelligent measure of "similarity" on the nodes so that these "similar" nodes coalesce. This implicit measure of similarity is closely linked to whether the nodes belong to the optimal partition or not. Furthermore, instead of using some classification method which stipulates the membership of the nodes into groups, the system adaptively decides the grouping by extracting information about the relative groupings of the various nodes when they are considered in pair-wise migrations. To achieve this, the algorithm uses previous sub-partition patterns to intelligently partition the entire node set so as to obtain a superior partitioning. As a by-product of the scheme, the solution not only decides the partitionings but also quantifies the "closeness of fit" of how *well* the nodes belong to the corresponding sub-partitions. To our knowledge, ours is the first scheme capable of quantifying this.

There are many advantages to this approach. Unlike search methods, the search for a better solution is not just for a new partitioning in the "neighbourhood" of the current one. Instead, pairs (and sometimes, a group of three/four) nodes are compared in a pair-wise manner so as to achieve the learning. Also, the technique is adaptive. Furthermore, unlike heuristic methods [Ni80,Ro91,RP92] which merely impose a search criterion for closeness between two partitions, we can now generalize a closeness criterion to extrapolate whether a node belongs to a potential sub-partition. Finally, (and far from being insignificant) the scheme also provides a mechanism by which we can nominate the "best" (or most representative) node for each sub-partition. This node which can be thought of as the *nucleus* of the respective sub-partition is automatically and adaptively stipulated by the system. The algorithm is both time and space inexpensive (sorts in place) and provides a good solution quickly which can subsequently be improved upon by using one of the other local improvement schemes which take care of the "straggler nodes". Finally, the automaton recommended here can also be adapted for the GPP in which the edge costs are not *constant* but random variables whose distributions are *unknown*. We believe that this is the first solution which is capable of this.

The strategy in this paper utilizes the philosophy of the Object Migrating Automaton that has been proven to be powerful in solving the Object Partitioning Problem (OPP) [YSL81,OM88]. The OPP can be defined as follows: Let $\mathbf{A} = \{A_1,...,A_W\}$ be a set of W objects to be partitioned into K classes $\{\pi_1,...,\pi_K\}$. The objects are partitioned such that those that are frequently accessed together lie in the same class. This implies that there is a correct partitioning, called a grouping, for the objects. In the most straightforward form of the OPP, objects are accessed in pairs $<A_i, A_j>$ called a "Query". It is assumed that the objects within the same grouping are accessed more frequently together than with objects in other groupings. Thus, the solution to the OPP will process a sequence of queries so as to increase the likelihood that the pair of objects accessed will reside in the same class the next time an identical query is formed.

Unlike the general case of object partitioning the GPP is closely related to the special case in which all the classes are of equal size. This problem is referred to as the Equi-Partitioning Problem (EPP). The best non-automaton solution to the EPP is the method due to Yu *et. al.* [YSL81]. In this method, initially each member object, A_i, is associated with a real number X_i. Whenever a query $<A_i, A_j>$ is processed, the quantities $\{X_i, X_j\}$ are moved towards their centroid by some amount δ_1. Then, a random pair $<X_p, X_q>$, $(1\leq p,q\leq W)$ is moved away from *its* centroid by some amount δ_2. The result is that similar objects will gradually cluster together and dissimilar objects will separate.

The best known solution for the EPP is the Object Migrating Automaton (OMA) proposed by Oommen and Ma [OM88] which has already been used in a number of application domains [OF93, OVZ92,OZ93]. The OMA moves all the W objects around within its states as opposed to traditional automata which simply move from one state to another. Thus, when applied to the EPP a solution is not defined by the current state of the OMA; instead it is defined by the entire structure of the automaton. The mapping function defining the transition of the automaton from one state to another essentially defines the motion of two or three objects within this structure. When a query $<A_i, A_j>$ is received, if A_i and A_j are in the same class, they are both rewarded and moved towards the most internal state associated with that class. This motion is executed one step at a time. However, if A_i and A_j are in different classes, they are moved towards the states sharing common boundaries with the other classes one step at a time, and whenever one of them is in the boundary state it is migrated towards the corresponding state of the other action.

The OMA is extremely accurate and fast. Experimentally, it converges to the true solution all the time, and does so in a speed which is an order of magnitude faster than the scheme due to Yu *et. al.* [YSL81] when all the objects are initialized to be in the boundary states of their respective classes (instead of initializing to random states). In the interest of brevity, we omit the detailed description of the OMA here and refer the reader to [OM88] for its structural details and for a review of the other reported solutions to equi-partitioning. We shall now adapt this philosophy for the GPP and demonstrate how efficient migration can be done to ensure an excellent partitioning.

IV. AUTOMATA-BASED GRAPH PARTITIONING

The input to our automaton, the Graph Partitioning Learning Automaton (GPLA) is the set of nodes $V=\{v_1,...v_{2N}\}$ to be partitioned into two equi-sized sub-partitions, V_1 and V_2 and the set of edge weights. The GPLA can be easily generalized for the case of partitioning V into K sub-partitions by designing it so as to possess K actions. The objects migrated by the GPLA are the conceptual set of nodes, V, with the aim that the nodes that are strongly connected and which hopefully belong to the partition choose the same action.

For the rest of this section we assume that the GPLA equi-partitions V into K subsets. Initially, all the objects (each of which represents a node) are placed randomly into the K actions. We define the Graph Partitioning Learning Automaton (GPLA) as a 7-tuple as below :

$(V, E , \{\phi_1, \phi_2, ..., \phi_{KM}\}, \{\alpha_1, \alpha_2, ..., \alpha_K\}, \beta, \mathbf{Q}, \mathbf{G})$, where,

(i) $V=\{v_1,...v_{KN}\}$ is the set of nodes to be partitioned.

(ii) E is the edges with the associated cost matrix C.

(iii) $\{\phi_1, \phi_2, ..., \phi_{KM}\}$ is the set of states. M is called the "Depth of Memory" of the machine.

(iv) $\{\alpha_1, \alpha_2, ..., \alpha_K\}$ is the set of K actions, each representing a certain sub-partition into which the elements of V must fall.

(v) $\beta = \{0, 1\}$ is its set of inputs where '1' represents a penalty and '0' a reward.

(vi) **Q**, the transition function specifies how the objects (the conceptual nodes) should move between the various states and is quite involved. It will be explained in detail presently.

(vii) The function **G** partitions the set of states for the sub-partitions. For each action α_j, there is a set of states $\{\phi_{(j-1)M+1}, ..., \phi_{jM}\}$, where M is the depth of memory. Thus,

$$\mathbf{G}(\phi_i) = \alpha_j \quad \text{if} \quad (j-1)M + 1 \le i \le jM \tag{5}$$

This means that the node in the automaton chooses α_1 if it is in any of the first M states, it chooses α_2 if it is in any of the states from ϕ_{M+1} to ϕ_{2M}, etc. We assume $\phi_{(j-1)M+1}$ to be the most internal state of action α_j, and ϕ_{jM} to be the boundary state. These are called the states of *MaximumCertainty* and *MinimumCertainty* respectively

As in the case of the OMA, we shall require that all the elements of V move among the states of the machine, and thus it is distinct from traditional learning automata. Also, if node v_i is in action α_j, it signifies that it is in the sub-partition whose index is j. Observe too that if the states occupied by the nodes are given, the sub-partitions can be trivially obtained using (5). This will completely specify the set of sub-partitions currently dictated by the GPLA.

Let $\omega_i(n)$ be the index of the state occupied by node v_i at the n^{th} time instant. Based on $\{\omega_i(n)\}$ and (5) let us suppose that the GPLA decides a current partitioning of V into sub-partitions. Using this notation we shall later describe the transition map of the GPLA.

Since the intention in the learning process is to collect "similar" nodes of the graph into the same sub-partition, the question of "inter-node similarity" (i.e., which two nodes should be grouped

together) is rather crucial. The philosophy we have used to quantify this is analogous to that used by Kernighan and Lin. We shall reckon that two nodes are "similar" if they are "strongly" connected, and thus their corresponding edge cost is small. Otherwise they will be dissimilar. Explicitly, we reckon that they are strongly connected if their edge cost is $(1+\rho)$ times the mean edge cost of the entire graph, where ρ is a user-specified parameter of the scheme. Analogously, they are deemed weakly connected if their edge cost is $(1-\rho)$ times the mean edge cost.

First of all, observe that the different states within a given sub-partition quantify the measure of certainty that the scheme has for a given node belonging to the sub-partition in question. At system start-up all the nodes are placed in the boundary state (of *MinimumCertainty*) of their initially randomly chosen sub-partitions indicating that the scheme is initially uncertain of the placement of all the nodes. As the learning proceeds, "similar" nodes of the graph will be rewarded for their being together in the same sub-partition and they will thus migrate towards their most internal state of the sub-partition -- their corresponding states of *MaximumCertainty*, indicating that the system is more certain that the nodes belong there. Likewise other nodes will be penalized and are either moved towards their boundary state or to another sub-partition, indicating the system's current ambiguity in associating them to the current sub-partition.

Initially, the GPLA has a preprocessing phase in which the mean edge cost is evaluated. The algorithm now moves into its main learning loop. The edges are repeatedly processed in a random order. Whenever an edge e_{ij} is processed, if v_i and v_j are "similar" and they currently belong to the same sub-partition, the automaton (and in particular, v_i and v_j) is rewarded. This mode of rewarding is called the *RewardSimilarNodes* mode. However, if they are both assigned to distinct sub-partitions, the automaton is penalized. This mode of penalizing is called the *PenalizeSimilarNodes* mode, because, in this mode, nodes which are actually similar are assigned to distinct sub-partitions, and the partitioning is therefore to be penalized.

As opposed to this, if v_i and v_j are "dissimilar" and they currently belong to the distinct sub-partitions, the automaton (and in particular, v_i and v_j) is rewarded. This mode of rewarding is called the *RewardDissimilarNodes*. However, if they are both assigned to the same sub-partition (which they should not be assigned to), the automaton is penalized. Analogous to the above, this mode of penalizing is called the *PenalizeDissimilarNodes* mode, because, in this mode, nodes which are actually dissimilar are assigned to the same sub-partition and they are therefore penalized.

The cycle then continues to the next iteration where the rewarding and penalizing phases repeat.

We now describe the actual transitions described by **Q** for each of these operations.

(i) Transitions for Rewards : RewardSimilarNodes

This is the case encountered when nodes v_i and v_j are reckoned to be similar and simultaneously in the same sub-partition. Since they are in the same sub-partition, say, α_k, both of

them are moved toward the most internal state of that sub-partition, $\phi_{(k-1)M+1}$, one step at a time. See Figure I(a). If either is in the most internal state, it merely remains in that state.

(ii) Transitions for Rewards : RewardDissimilarNodes

In this case v_i and v_j are reckoned to be dissimilar and simultaneously in distinct sub-partitions say, α_k and α_m respectively. This being a favourable scenario, both the nodes are moved towards the most internal state of that sub-partition, $\phi_{(k-1)M+1}$ and $\phi_{(m-1)M+1}$ one step at a time. As in (i) above, if either is in the most internal state of *its* action, it merely remains in that state.

(iii) Transitions for Penalties : PenalizeSimilarNodes Mode

This is the case encountered when two similar nodes, v_i and v_j, are located in distinct sub-partitions. Let us assume that v_i and v_j lie in different sub-partitions, say α_k and α_m respectively, (i.e. v_i is in state ω_i, where $\omega_i \in \{\phi_{(k-1)M+1},.., \phi_{kM}\}$, and v_j is in state ω_j, where $\omega_j \in \{\phi_{(m-1)M+1}, ..., \phi_{mM}\}$). Then they are moved away from $\phi_{(k-1)M+1}$ and $\phi_{(m-1)M+1}$ as follows:

a) If $\omega_i \neq \phi_{kM}$ and $\omega_j \neq \phi_{mM}$, then move v_i and v_j one state towards ϕ_{kM} and ϕ_{mM} respectively. We thus move them towards the boundary states. See Figure I(b).

b) If at least one of v_i or v_j is in the boundary state of *MinimumCertainty*, (i.e. either $\omega_i = \phi_{kM}$ or $\omega_j = \phi_{mM}$) , then move the node in the boundary state, say v_i, to ϕ_{mM}, the boundary state of α_m. In this case, since this will result in an excess of nodes in the partition represented by α_m, one of the nodes in α_m other than v_j is moved to ϕ_{kM}, the boundary state of α_j. We choose to move the one closest to ϕ_{mM}. See Figure I(c).

(iv) Transitions for Penalties : PenalizeDissimilarNodes Mode

This case is encountered when two dissimilar nodes, v_i and v_j, are simultaneously located in the same sub-partition, say, α_k. They are then both moved away from $\phi_{(k-1)M+1}$ as follows:

a) If $\omega_i \neq \phi_{kM}$ and $\omega_j \neq \phi_{kM}$, then move both v_i and v_j one state towards ϕ_{kM} the boundary states. See Figure I(d).

b) If at least one of v_i or v_j is in the boundary state of *MinimumCertainty*, (i.e. either $\omega_i = \phi_{kM}$ or $\omega_j = \phi_{kM}$) , then move the node in the boundary state, say v_i, to ϕ_{ZM}, the boundary state of α_Z. In the 2-way GPP the latter action is merely the partition which they are **not** currently in. However, in the K-way GPP the migrated node is repeatedly moved to all the other K-1 partitions and finally stationed in the best of these, which is the one which minimizes the inter-set costs. Again, since this will result in an excess of nodes in the partition represented by α_Z, one of the nodes in α_Z closest to ϕ_{ZM} is moved to ϕ_{kM}. See Figure I(e).

The GPLA has an excellent partitioning capability. Typically, it yields an excellent initial partitioning in an order of magnitude faster than all the other algorithms, Thereafter a single iteration of a local search algorithm (either Algorithm XLS or Algorithm KL) yields our final solution. The combination of the two is referred to as Algorithm GP_System formally given below.

Algorithm GP_System
Input : The set V = $\{v_1,...v_{KN}\}$, to be partitioned into K sub-partitions.
Output : The final solution of the GPP
Assumption: The algorithm can invoke the GPLA and either KL or XLS
Method
$\{V_1, V_2, ..., V_K\}$:= GPLA(V)
Final _Solution := Do_Local_Search($V_1, V_2, ..., V_K$) /* Do One iteration of KL or XLS*/
End Algorithm GP_System

Algorithm GPLA

Input : The set V = $\{v_1,...v_{KN}\}$, to be partitioned into K sub-partitions.
C is the adjacency cost matrix and $V_1, V_2, ..., V_K$ are current feasible sub-partitions.
ρ, a parameter used to decide whether two nodes are reckoned "similar", i.e., strongly connected. In our implementation ρ:= 0.25.
M is the number of states associated with a class. In our experiments M := 10.
The algorithm is run for a fixed number of iterations. In our implementation the number of iterations was K·100·N, where K·N is the size of the node set[6].
Output : The final partitions $\{V_1, V_2, ..., V_K\}$.
Notation : ω_i is the state of the node v_i. It is an integer in [1..KM], where,
if (j-1)M +1 ≤ ω_i ≤ jM, then node v_i is assigned to the sub-partition α_j.

PreProcess
Compute Mean_Edge_Cost
Randomly create K equi-sized subsets of V, $\{V_1, V_2, ..., V_K\}$ and assign the nodes to the boundary states of the corresponding actions.

Method
For Iteration := 1 **to** Max_Iterations **Do**
For a random edge E_{ij} **Do**
If C_{ij} > (1+ ρ) ·Mean_Edge_Cost **Then**
If v_i and v_j are in the same class **Then**
RewardSimilarNodes (i,j)
Else
PenalizeSimilarNodes (i,j)
EndIf
Else
If C_{ij} < (1- ρ) ·Mean_Edge_Cost **Then**
If v_i and v_j are in the same class **Then**
PenalizeDissimilarNodes (i,j)
Else
RewardDissimilarNodes (i,j)[7]
EndIf
EndIf
EndIf
EndFor
EndFor
Return the final partitions $\{V_1, V_2, ..., V_K\}$
End Algorithm GPLA

---

[6] In some cases the number of iterations was only K·50·N, where K·N is the size of the node set. See Section IV for details.
[7] Procedures *RewardSimilarNodes* and *RewardDissimilarNodes* are identical; they are written separately only to clarify the different modes of the algorithm.

Procedure RewardSimilarNodes (i,j)
Input : Node indices i and j where ω_i and ω_j are the state indices of similar nodes.
v_i and v_j are in the same sub-partition.
Method
If $\omega_i \bmod M \neq 1$ **Then** /* i is not in the most internal state */
$\omega_i := \omega_i - 1$
If $\omega_j \bmod M \neq 1$ **Then** /* j is not in the most internal state */
$\omega_j := \omega_j - 1$
End Procedure RewardSimilarNodes

Procedure RewardDissimilarNodes (i,j)
Input : Node indices i and j where ω_i and ω_j are the state indices of dissimilar nodes.
v_i and v_j are in distinct sub-partitions.
Method
If $\omega_i \bmod M \neq 1$ **Then** /* i is not in the most internal state */
$\omega_i := \omega_i - 1$
If $\omega_j \bmod M \neq 1$ **Then** /* j is not in the most internal state */
$\omega_j := \omega_j - 1$
End Procedure RewardSimilarNodes

Procedure PenalizeSimilarNodes (i,j)
Input : Node indices i and j where ω_i and ω_j are the state indices of similar nodes.
v_i and v_j are in distinct sub-partitions.
Method
If $((\omega_i \bmod M) \neq 0)$ **and** $((\omega_j \bmod M) \neq 0)$ **Then** /* Both are in internal states */
$\omega_i := \omega_i + 1$
$\omega_j := \omega_j + 1$
Else
If $(\omega_i \bmod M \neq 0)$ **Then** /* v_i is in an internal state */
$\omega_i := \omega_i + 1$ /* Update state of v_i */
temp := ω_j /* Store the state of v_j */
$\omega_j := (\omega_i \text{ DIV } M) * M$ /* Move v_j to same group as v_i */
t := index of a node in sub-partition of v_i
where $v_t \neq v_i$ and is closest to boundary state of ω_i
ω_t := temp /* Move v_t to the old state of v_j */
Else
If $(\omega_j \bmod M) \neq 0)$ **Then** /* v_i has to be moved */
$\omega_j := \omega_j + 1$ /* Update state of v_j */
temp := ω_i /* Store the state of v_i */
$\omega_i := (\omega_j \text{ DIV } M) * M$ /* Move v_i to same group as v_j */
t := index of a node in sub-partition of v_j
where $v_t \neq v_j$ and is closest to boundary state of ω_j
ω_t := temp /* Move v_t to the old state of v_i */
EndIf
EndIf
EndIf
End Procedure PenalizeSimilarNodes

Procedure PenalizeDissimilarNodes (i,j)
Input : Node indices i and j where ω_i and ω_j are the state indices of dissimilar nodes.
v_i and v_j are in the same sub-partition.
Method
If ((ω_i mod M) ≠ 0) **and** ((ω_j mod M) ≠ 0)) **Then** /* Both are in internal states */
$\omega_i := \omega_i + 1$
$\omega_j := \omega_j + 1$
Else
If (ω_i mod M ≠ 0) **Then** /* v_i is in an internal state */
$\omega_i := \omega_i + 1$ /* Update state of v_i */
TempState1 := ω_j /* Store the state of v_j */
Solution_Cost := EvaluateCost of current partitioning
For all the remaining K-1 partitions **Do**
ω_P := state of node closest to boundary in this current sub-partition
TempState2 := ω_P
$\omega_j := (\omega_P \text{ div } M + 1) * M$ /* Move v_j to new sub-partition */
ω_P := TempState1 /* Move v_P to the old state of v_j */
New_Cost := EvaluateCost of current partitioning
If New_Cost > Solution_Cost **Then** /* This change is not superior */
ω_P := TempState2 /* Undo it */
ω_j := TempState1
Else /* This change is superior */
Solution_Cost := New_Cost /* Retain it */
EndIf
EndFor
Else /* v_j is in an internal state */
$\omega_j := \omega_j + 1$ /* Update state of v_j */
TempState1 := ω_i /* Store the state of v_i */
Solution_Cost := EvaluateCost of current partitioning
For all the remaining K-1 partitions **Do**
ω_P := state of node closest to boundary in this current sub-partition, α_Z
TempState2 := ω_P
$\omega_i := (\omega_P \text{ div } M + 1) * M$ /* Move v_i to new sub-partition */
ω_P := TempState1 /* Move v_P to the old state of v_i */
New_Cost := EvaluateCost of current partitioning
If New_Cost > Solution_Cost **Then** /* This change is not superior */
ω_P := TempState2 /* Undo it */
ω_i := TempState1
Else /* This change is superior */
Solution_Cost := New_Cost /* Retain it */
EndIf /* Move v_t to the old state of v_i */
EndFor
EndIf
EndIf
End Procedure PenalizeSimilarNodes

Note that although the fundamental principles involved in the individual migrations are based on the philosophy used in the OMA (namely, the Tsetlin-like[8] transitions on being rewarded and penalized), the algorithm is completely different. The primary differences are the following :

[8]Tsetlin-like transitions are transitions which go one step deeper into the most internal state of an action on being rewarded. They unlearn by going towards the peripheral states on being penalized, one step at a time. This sort of learning was initially motivated by the Linear Tactic proposed by Tsetlin [TS73,NT89].

(i) Unlike the OMA where the migrations are done "on request" (i.e., when a user performs a query), in the GPLA the migrations are performed for all pairs of nodes processed. Furthermore, these node pairs are processed randomly.

(ii) Unlike the OMA, which has no way of penalizing "non-accessed elements" the GPLA has a strategy of penalizing them by considering how similar the nodes within the same sub-partition are. Clearly, this cannot be done in the OMA because, in that case, the system is absolutely dependent on the users' queries. In this case the system can quantify how fitting a node is for a sub-partition, because the internal state of a node is an indication of the latter.

(iii) Unlike the OMA, the GPLA permits us to quantify how suitable a node is for a sub-partition. This is analogous to the technique used in string taxonomy [OD94] and to the distance metrics used in *statistical* pattern recognition where the mean feature for a class can serve as its representative. In this case, although such a mean does not exist, the node closest to the most internal state can be reckoned to be the node that best represents *that* sub-partition. This is the primary advantage of our scheme when compared to the other reported algorithms. Like the other algorithms discussed in Section II, we are able to make perturbations to the current partitioning. However, unlike the other algorithms, we do not merely represent the solution as sets. Rather, each node in the solution sets also have an index of how certain the algorithm is in assigning it to the set. Thus, perturbations to a current solution are made to involve only the nodes closest to the boundaries -- the nodes whose assignments the algorithm is least certain of. This is probably the reason for the excellent performance of the GPLA.

(iv) Finally, the concept of selectively processing nodes using ρ is unique to the GPLA. Unlike the OMA, we are not dependent on the users' query patterns. Thus, once the graph is available, the edges are randomly chosen and the corresponding nodes are either rewarded or penalized depending on their current locations in the sub-partitions. But, whenever an edge cost is greater than $(1+ \rho)$ times the mean edge cost we treat the nodes to be strongly connected. Similarly, whenever an edge cost is less than $(1- \rho)$ times the mean edge cost we treat the nodes to be weakly connected. The edges are thus selectively processed, and the domain between these upper and lower bounds is reckoned as an "unsafe" zone where the decisions of partitioning have to be made very conservatively.

V. EXPERIMENTAL RESULTS

The GPLA has been rigorously tested for the 2-way GPP and the results that we have obtained are quite fascinating. The data which was used was obtained from four types of graphs :

(i) Dense graphs with Euclidean distances as edge costs

(ii) Sparse graphs with Euclidean distances as edge costs

(iii) Dense graphs with random edge costs

(iv) Sparse graphs with random edge costs.

The Euclidean graphs were generated by randomly selecting real-valued points in a 100 x 100 Cartesian square. As opposed to this, the random edge-cost graphs were obtained by uniformly generating real numbers chosen from the interval [0, 100]. Also, to create a sparse graph, each edge was deleted with an independent probability of 0.33. For each of the above graph types the cardinality of the node set was varied in the various experiments. Also, to ensure that the experiments were meaningful, a depth first search was run on each generated input graph so as to prevent disconnected graphs from being used for any test. Finally, to ensure uniformity in reporting the results, all the algorithms examined were tested for the same graphs.

As mentioned earlier, although the GPLA finds fast solutions these are rarely the global optima. This is probably due to the fact that it is incapable of detecting "straggler nodes" (as Kernighan calls them). These stragglers do not know which is the best class they should belong to and consequently tend to migrate back and forth in the various sub-partitions, and thus tend to stay in the corresponding boundary states. Thus, the GPLA demonstrates a marked improvement if it was followed by a single local search iteration using either the Kernighan-Lin scheme or Extended Local Search. In other words the GPLA finds the region close to an optimal solution quickly.

Experimentally, the structure of the graph also played a major role in determining which algorithm would find the best solution. Thus, since sparse graphs contained deeper local optima and fewer good solutions, they were excellent candidates for the GPLA. Also, as reported in [PR94,Ro91,RP92] for smaller dense graphs the Algorithm KL demonstrated an excellent performance. In the interest of comparison, Algorithm KL was run for a single iteration and for ten iterations (referred to as Algorithms KLx1 and KLx10) respectively. Unfortunately, the Extended Genetic Algorithm was very slow and so, in our implementation, the number of generations considered was bounded[9] by 10. However, given enough time it is possible that (as reported in [Ro91, RP92]) both Algorithms XLS and XGA would have found better solutions than those reported here.

The three automaton solutions tested were :

(i) The GPLA scheme with $\rho := 0.25$.

(ii) The GPLA scheme with $\rho := 0.25$ augmented with *one* iteration of Algorithm XLS (GP_XLS).

(iii) The GPLA scheme with $\rho := 0.25$ augmented with *one* iteration of Algorithm KL (GP_KL).

In case (i) the number of iterations done was 100 times the number of actual edges in the graph, **G**, and in cases (ii) and (iii) the number of iterations done was only 50 times the number of edges in **G**.

The actual test suite consisted of five sets of experiments. The details are listed in Table I below:

[9]One referee had suggested that we increase the number of generations to 100. On attempting to do this we observed that the time taken for the scheme was prohibitively large. Indeed, running the XGA for 10 generations itself was a computationally colossal exercise -- for larger graphs the algorithm did not even terminate after running all night on a Pentium 75. In terms of computational efficiency, increasing the number of generations would only render the XGA even less suitable as a competitor for solving the GPP.

Experiment No.	No. of Runs	No. of Nodes	Methods Tried
Experiment 1	1	10-80	XLS, XGA,KL,GLPA,GP_XLS,GP_KL
Experiment 2	3	20-160	KLx1,KLx10,XGA,GPLA,GP_KL
Experiment 3	10	20-180	KLx1,KLx10,GPLA,GP_KL
Experiment 4	2	100-380	KLx1,KLx10,GPLA,GP_KL
Experiment 5	1	500	KLx1,KLx10,GPLA,GP_KL

Table I : Overview of the experiments conducted

Experiment Set 1

In these experiments to measure how the various schemes would perform on *individual* runs, the various algorithms were tested for sample individual graphs on an IBM-486 running on the MACH operating system. The node set was varied from 20 to 80, and in each case, the convergence cost and the time required to converge was recorded for all the four types of graphs.

The results that we have obtained are remarkable (See Table II). In terms of accuracy, all of the schemes are comparable. But when it concerns time, the last of the above three (GP_KL) was the most superior. Thus, for example, for a complete random graph with 80 nodes, all the algorithms converged to solutions possessing almost the same accuracy. However, Algorithm XLS required approximately 1078 seconds, and Algorithm XGA took 518 seconds. As opposed to this, Algorithm KL required only about 111 seconds. The automaton solutions were much faster. The GPLA by itself required about 47 seconds when the number of iterations done was $100 \cdot N_{edges}$. Augmented with one iteration of Algorithm KL the system (with $50 \cdot N_{edges}$ iterations) required only about 24.5 seconds. For the sparse Euclidean graph with 80 nodes all the algorithms were again almost as accurate. But whereas, Algorithm XLS required approximately 1116 seconds, Algorithm XGA took 530 seconds and Algorithm KL required only about 207 seconds. The automaton solutions were again superior. The GPLA (when the number of iterations done was $100 \cdot N_{edges}$) by itself required about 49 seconds. Augmented with a single iteration of Algorithm KL the system (with $50 \cdot N_{edges}$ iterations) required only about 25.6 seconds. Such results, implying an order of magnitude superiority are typical.

Experiment Sets 2-5

In these sets of experiments, the performance of the algorithms were tested for *average* behaviour. In each case, the results reported are the average costs obtained and times required for all the respective types of graphs when the algorithms were run on a Pentium 75 running on Windows 3.11. Since the XGA required enormous computations, in Experiment 2 (See Table III) the number of nodes were varied from 20 to 160, and in each case only three runs were conducted. In Experiment 3, since the XGA was not included in the testing, ten experiments were conducted to do the averaging (See Table IV). In Experiment 4, (See Table V) we studied graphs with large sets of nodes (up to 380), and since the times involved did not permit us to do too many parallel runs, only

two runs were done for each graph type. Finally, in the last set of experiments we investigated the behaviour of the algorithms for graphs of 500 nodes (See Table VI).

In all cases the results obtained are quite conclusive. Thus, (See Table III) for a complete random graph with 140 nodes, all the algorithms converged to solutions possessing almost the same accuracy. However, Algorithm XGA took 381 seconds. As opposed to this, Algorithm KL required only about 9.7 seconds for a single iteration, and about 103.0 seconds for ten iterations. The automaton solutions were much faster. The GPLA by itself required only about 2.7 seconds, but augmented with a single iteration of Algorithm KL the system needed only about 12.3 seconds. Consider now (See Table IV) the case of a sparse Euclidean graph with 140 nodes. Here too all the algorithms converged to solutions possessing almost the same accuracy. But whereas, a single iteration of Algorithm KL required only about 11.40 seconds and ten iterations of the latter took 117.10, the GPLA by itself required about 2.50 seconds. Enhanced with an iteration of Algorithm KL the system required an average of 17.3 seconds -- an order of magnitude less computation. Similar advantages are seen in the results tabulated in Tables V and VI for graphs of larger size. In some cases, the GPLA alone gave an exceptional performance. Thus, for a sparse random graph of 500 nodes, the GPLA working on a standalone basis converged to an excellent value (to a value which is less than 0.2 % of the terminal value given by Algorithm KLx10 !!!) in less than 2.5 % of the time required by the latter. Indeed, in all cases, the power of our new system is obvious, and its superiority over the current methods seems to increase with the size of the graph (See Figure I).

VI. CONCLUSIONS

In this paper we have presented, to our knowledge, the first reported Learning Automaton (LA) solution to the NP-Complete Graph Partitioning Problem (GPP) which involves partitioning the nodes of a graph **G** into two (in general, K) sets of equal size so as to minimize the sum of the costs of the edges having end-points in different sets. We have compared our solution to various reported schemes such as the Kernighan-Lin's algorithm, and two excellent recent heuristic methods proposed by Rolland *et. al.* The current automaton-based algorithm outperforms all the other schemes with regard to computation time -- we believe that it is the fastest algorithm reported to date.

We are currently investigating using LA for the GPP when the edge costs are random variables with *unknown* distributions. Also, just prior to the revision of this paper we were sent a new unpublished manuscript containing results on using the Tabu Search for the GPP. The latter results are extremely impressive and seem to be comparable, in terms of computational efficiency, to the methods developed here. We are currently attempting to study these two methods comparatively.

Acknowledgments : We are especially grateful to the anonymous referees who painstakingly reviewed the paper and provided us with various comments that improved its quality.

REFERENCES

[AK89] Aarts, E. and Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley and Sons, New York, 1989.

[CDQ91] Cox Jr., L. A., Davis, L. and Qiu, Y., "Dynamic Anticipatory Routing in Circuit-Switched Telecommunications Networks", *Handbook of Genetic Algorithms* (Ed. by L. Davis), Van Nostrand Reinhold, New York, 1991.

[DS89] De Jong, K. A. and Spears, W.M., "Using Genetic Algorithms to Solve NP-Complete Problems", *Proc. of the 3rd International Conf. on Genetic Algorithms* (J. D. Schaffer, ed.). George Mason University, 1989.

[FK90] Feo, T.A. and Khellaf, M., "A Class of Bounded Approximation Algorithms for Graph Partitioning", *Networks*, 20, 181-195, 1990.

[GJ79] Garey, M. R. and Johnson, D.S., *Computers and intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[GJS76] Garey, M. R., Johnson, D.S. and Stockmeyer, L., "Some Simplified NP-Complete Graph Problems", *Theor. Comput. Sci.*, 1, 237-267, 1976.

[Go87] Goldberg, D. E., "Computer-Aided Pipeline Operation Using Genetic Algorithms and Rule Learning. Part 1: Genetic Algorithms in Pipeline Optimization", *Engineering with Computers*, 3, 35-45, 1987.

[Go89] Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company, Reading, MA., 1989.

[GS86] Golden, B. L. and Skiscim, C.C., "Using Simulated Annealing to Solve Routing and Location Problems", *Naval Research Logistics Quarterly*, 33, 261-279, 1986.

[JAM89] Johnson, D. S., Aragon, C.R., McGeoch, L.A. and Schevon, C., "Optimization by Simulated Annealing: An Experimental Evaluation; Part 1, Graph Partitioning", *Operations Research*, 37, 6, 1989.

[KL70] Kernighan, B.W. and Lin, S., "An Efficient Heuristic Procedure For Partitioning Graphs". *The Bell Systems Technical Journal*, 49, 291-307, 1970.

[LH89] Liepins, G. E. and Hilliard, M.R., "Genetic Algorithms: Foundations and Applications", *Annals of Operations Research*, 21, 31-58, 1989.

[Ni80] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga Publishing, Palo Alto, CA, 1980.

[NT89] Narendra, K.S. and Thathachar, M.A.L., *Learning Automata : An Introduction*, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.

[OD94] Oommen, B.J. and De St. Croix, T., "String Taxonomy Using Object Migrating Automata". To appear in the *IEEE Transactions on Systems, Man and Cybernetics*.

[OF93] Oommen, B.J. and Fothergill, C., "Fast Learning Automaton-Based Image Examination and Retrieval", *The Computer Journal*, Vol. 36, No. 6, 1993, pp. 542-553.

[OM88] Oommen, B.J. and Ma, D.C.Y., "Deterministic Learning Automata Solutions to the Equipartitioning Problem", *IEEE Transactions on Computers*, Vol. 37, 1988, pp. 2-13.

[OVZ92] Oommen, B.J., Valiveti, R.S. and Zgierski, J., "An Adaptive Learning Solution to the Keyboard Optimization Problem", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-22, September/October 1992, pp. 1233-1243.

[OZ93] Oommen, B.J. and Zgierski, J., "A Learning Automaton Solution to Breaking Substitution Cyphers", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-15, February 1993, pp. 185-192.

[PR94] Pirkul, H. and Rolland, E., "New Heuristic Solution Procedures for the Uniform Graph Partitioning Problem : Extensions and Evaluation". *Computers and Operations Research*, Oct 1994.

[Ro91] Rolland, E., *Abstract Heuristic Search methods for Graph Partitioning*, Ph.D. dissertation, The Ohio State University, Columbus, Ohio. 1991.

[RP92] Rolland, E. and Pirkul, H., "Heuristic Solution Procedures for the Graph Partitioning Problem", *Proc. of the 1992 ORSA-CSTS Conf. on Computer Science and Operations Research : New Developments in Their Interfaces*, Williamsburg, 1992, pp. 475-490.

[Ts73] Tsetlin, M.L., "Automaton Theory and the Modelling of Biological Systems", *New York and London, Academic*, 1973.

[YSL81] Yu, C.T., Siu, M.D., Lam, D. and Tai, T., "Adaptive Clustering Schemes: General Framework" in *Proc. of the IEEE COMPSAC Conf.*, 1981, pp. 81-89.

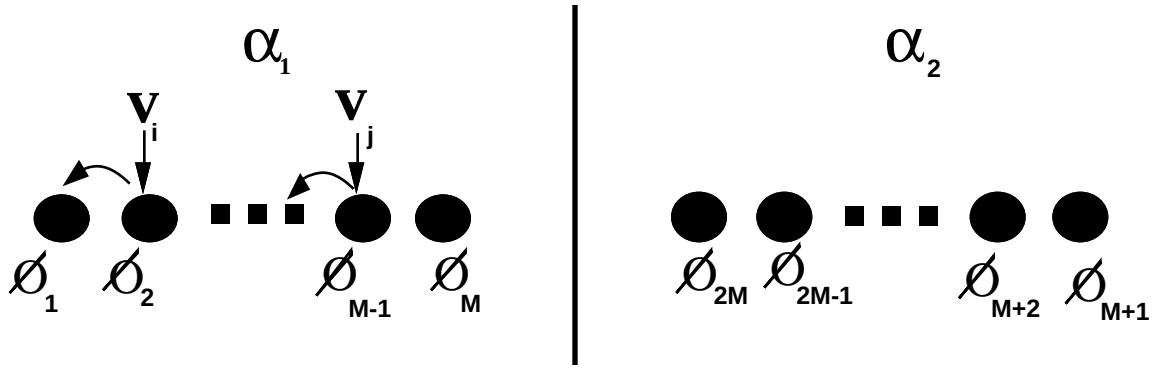

Figure I(a): Reward transitions for the 2M-State GPLA. In this case, (the *RewardSimilarNodes* mode) v_i and v_j are similar and located in the same sub-partition. If v_i and v_j were dissimilar (the *RewardDissimilarNodes* mode) and in distinct sub-partitions the transitions would be identical -- the nodes would move towards the most internal states of *the corresponding* sub-partitions one step at a time.

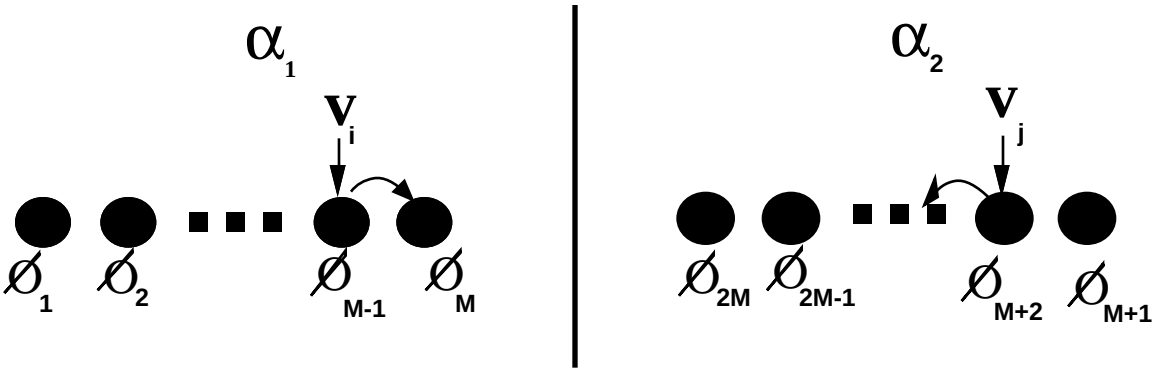

Figure I(b): Penalty transitions for the 2M-State GPLA -- *PenalizeSimilarNodes* Mode. v_i and v_j are similar but located in the distinct sub-partitions. Neither is in a boundary state.

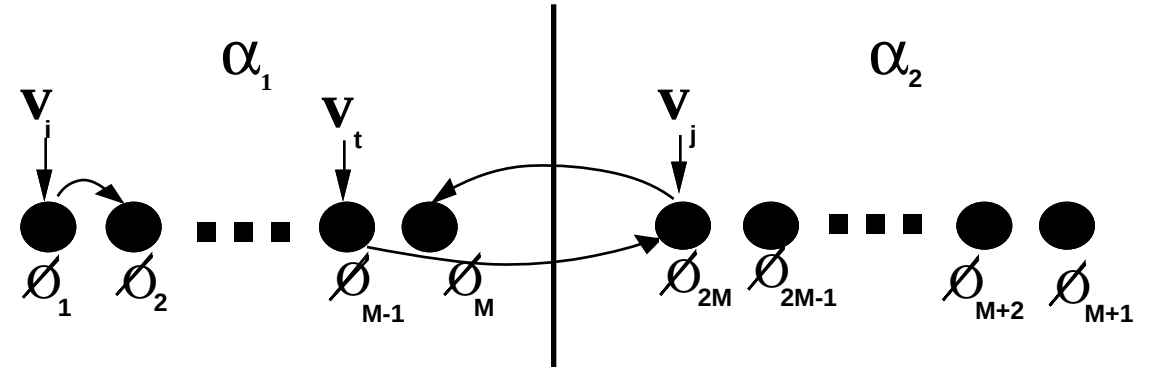

Figure I(c): Penalty transitions for the 2M-State GPLA -- *PenalizeSimilarNodes* Mode. Here v_i and v_j are similar but located in the distinct sub-partitions. However one of them (v_j) is in a boundary state.

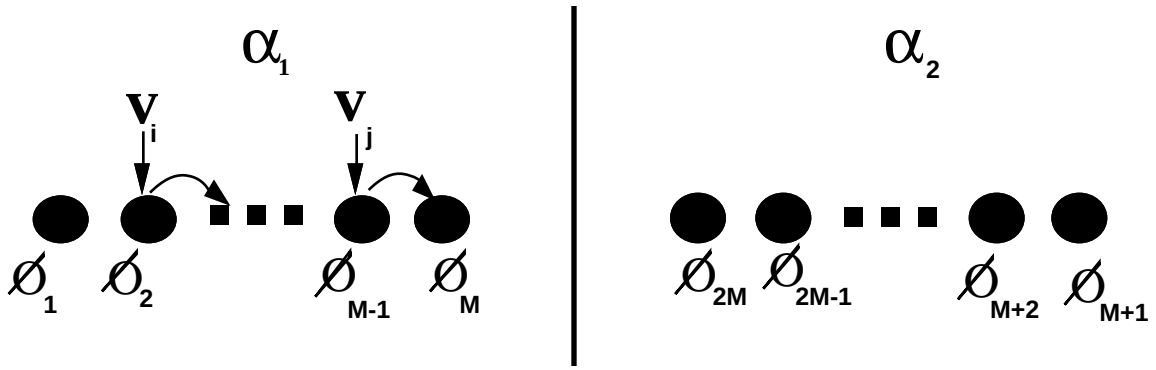

Figure I(d): Penalty transitions for the 2M-State GPLA -- *PenalizeDissimilarNodes* Mode. Here v_i and v_j are dissimilar but in the same sub-partition. Neither is in a boundary state.

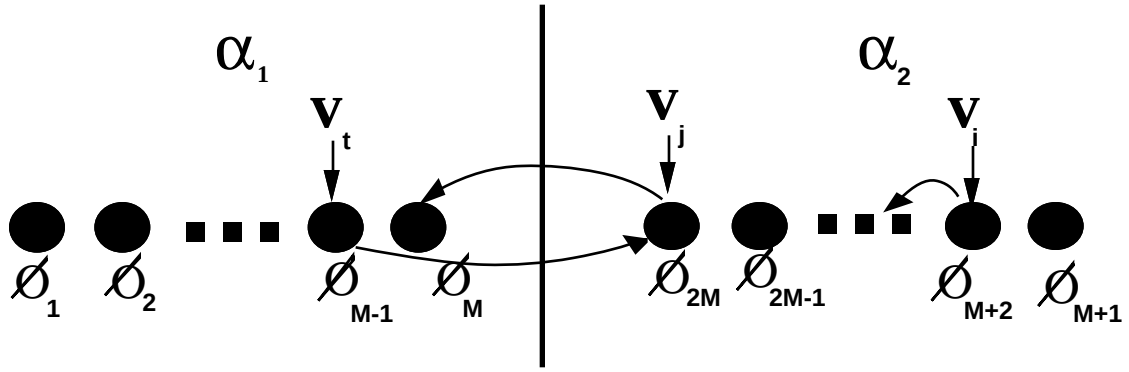

Figure I(e): Penalty transitions for the 2M-State GPLA -- *PenalizeDissimilarNodes* Mode. Here v_i and v_j are dissimilar but located in the same sub-partitions. However one of them (v_j) is in a boundary state.

Nodes	Sol./Time for XLS	Sol./Time for GPLA	Sol./Time for GP_XLS	Sol./Time for GP_KL	Sol./Time for XGA	Sol./Time for KL
10	1253/0.28	1280/0.83	1253/0.22	1253/0.44	1256/3.93	1253/0.30
20	4713/3.97	4749/2.89	4713/0.84	4713/1.67	4713/10.88	4713/4.37
30	10716/18.88	10788/6.34	10708/2.85	10711/3.46	10715/28.32	10708/9.85
40	19822/66.85	19947/11.48	19816/10.02	19809/6.28	19825/58.53	19817/41.92
50	33365/160.56	33650/18.39	33367/23.60	33361/9.56	33370/118.28	33370/111.61
60	48493/335.08	49000/26.46	48484/59.53	48483/14.15	48506/199.37	48485/215.26
70	64245/668.35	64896/36.24	64243/123.10	64233/21.04	64242/333.53	64230/413.97
80	84398/1183.00	84919/47.44	84392/265.78	84396/26.99	84409/509.71	84391/585.37

Table II(a) : Results for dense Euclidean graphs.

Nodes	Sol./Time for XLS	Sol./Time for GPLA	Sol./Time for GP_XLS	Sol./Time for GP_KL	Sol./Time for XGA	Sol./Time for KL
10	1085/0.54	1085/0.89	1085/0.39	1085/0.58	1085/4.22	1085/0.29
20	4385/3.96	4423/2.91	4360/0.65	4360/1.50	4423/10.19	4385/1.80
30	9881/19.55	10078/6.44	10020/1.83	10018/3.37	10018/25.05	9881/6.70
40	17778/64.30	18050/11.46	17872/4.13	17803/6.01	17767/59.16	17767/11.41
50	28587/156.50	28832/17.92	28548/9.74	28457/9.39	28157/114.92	28281/28.49
60	41090/335.00	41751/26.19	40759/28.20	41138/13.55	41365/209.10	41287/48.30
70	55945/630.28	56976/35.70	56008/56.53	55561/18.81	56119/332.02	56030/95.54
80	73274/1077.74	74766/46.61	74043/72.46	73190/24.51	74153/517.74	73274/110.92

Table II(b) : Results for complete random graphs.

Nodes	Sol./Time for XLS	Sol./Time for GPLA	Sol./Time for GP_XLS	Sol./Time for GP_KL	Sol./Time for XGA	Sol./Time for KL
10	747/0.29	726/0.92	653/0.13	653/0.52	653/4.01	653/0.48
20	2514/4.09	2514/2.95	2514/0.64	2514/1.68	2514/10.73	2514/2.00
30	5643/20.22	6094/6.51	5643/1.87	5643/3.45	5643/25.13	5725/8.86
40	11352/65.93	11632/11.58	11178/4.77	11071/6.65	11226/55.85	11263/23.68
50	17673/161.19	18066/18.33	18024/9.79	17795/9.57	18009/118.54	17628/60.17
60	26155/342.81	26397/26.77	26867/11.04	25909/13.89	26536/206.05	26581/76.58
70	37309/638.00	38320/37.05	37326/50.48	36923/20.10	37270/337.82	37114/100.62
80	48951/1116.45	50019/49.20	49180/151.52	48956/25.63	49082/530.00	48799/207.56

Table II(c) : Results for sparse Euclidean graphs.

Nodes	Sol./Time for XLS	Sol./Time for GPLA	Sol./Time for GP_XLS	Sol./Time for GP_KL	Sol./Time for XGA	Sol./Time for KL
10	680/0.28	780/0.93	680/0.22	680/0.36	680/4.18	680/0.22
20	2535/4.05	2638/3.00	2638/0.66	2638/1.68	2638/11.02	2535/1.19
30	6249/19.65	6196/6.66	6198/2.28	5871/3.52	5871/25.92	6352/7.95
40	11199/64.11	10837/12.94	11034/4.97	11000/6.11	10961/57.64	11037/19.92
50	17247/162.99	18088/18.38	17309/16.84	17155/9.66	17515/122.68	17247/37.77
60	24970/363.04	26223/26.91	25567/25.26	25838/13.94	25661/210.08	25862/51.67
70	35141/655.00	36053/36.46	34759/57.18	34673/19.50	34823/334.47	34813/99.85
80	46366/1160.00	47209/48.32	45994/134.38	45892/25.91	46313/509.76	46093/293.26

Table II(d) : Results for sparse random graphs.

Table II : Solution values and time required for individual runs of different GPP solutions for various graphs. The experiments were conducted on a IBM-486 machine running on the MACH operating system.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10	Sol./Time for XGA
20	5134/0.00	5049/0.00	5049/0.00	5049/0.00	5055/1.00
40	20871/0.00	20467/0.33	20463/0.67	20462/4.33	20476/7.33
60	49898/0.67	46781/2.67	45721/2.00	45718/12.33	45759/24.67
80	90525/1.00	88279/1.67	82140/6.00	82138/30.67	82214/59.00
100	141890/1.00	139188/2.00	127584/6.67	127446/64.33	127527/137.67
120	199169/1.33	195384/4.00	174892/26.67	174890/120.67	174996/231.00
140	277940/2.33	273561/6.67	242640/10.33	242428/140.67	242539/383.00
160	380258/3.00	375121/8.33	325054/31.33	324943/245.67	325138/580.67

Table III(a) : Results for dense Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10	Sol./Time for XGA
20	4313/0.00	4142/0.00	4111/0.00	4101/0.33	4174/0.67
40	18122/0.00	17643/0.00	17481/0.00	17429/2.00	17734/7.33
60	40146/0.67	39572/0.67	39340/0.67	39018/7.33	39983/26.00
80	72416/0.67	71586/2.00	71527/2.33	71250/17.00	72528/63.00
100	114283/1.00	113108/3.33	113905/3.00	112442/35.33	114736/139.33
120	165097/1.33	163759/6.33	163900/5.67	163207/70.00	167019/245.33
140	227039/2.67	223505/12.33	224631/9.67	223055/103.33	228213/380.67
160	296625/3.33	293525/13.33	294013/20.67	293373/149.00	299110/611.00

Table III(b) : Results for complete random graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10	Sol./Time for XGA
20	2923/0.00	2750/0.00	2715/0.00	2701/0.00	2743/1.00
40	11767/0.00	11160/0.00	11092/0.67	10952/2.00	11203/6.67
60	26691/0.67	25942/1.00	26337/1.00	25870/7.33	26413/25.33
80	58095/1.00	50003/2.67	49841/2.33	49670/18.67	51507/62.67
100	95273/1.00	78490/5.67	79020/3.33	77933/35.33	80493/134.67
120	129512/2.00	110691/9.67	112091/6.67	110075/68.33	113629/236.67
140	174515/3.00	151581/15.00	151865/11.33	151127/110.67	154717/377.00
160	252357/3.33	206561/24.00	207942/15.67	206272/162.00	211226/577.67

Table III(c) : Results for sparse Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10	Sol./Time for XGA
20	2490/0.00	2413/0.00	2393/0.00	2383/0.33	2474/1.00
40	11044/0.00	10515/0.33	10432/0.33	10400/2.00	10830/7.00
60	24691/0.67	24425/0.67	24036/1.00	23755/7.33	25023/25.67
80	46794/0.33	45673/2.00	45845/1.33	45268/18.33	47022/64.33
100	74018/1.00	72221/3.67	72327/3.33	71285/34.00	73951/135.00
120	108766/2.00	102965/7.33	103689/5.33	102198/59.00	106024/230.00
140	155549/3.00	142674/12.67	143084/9.00	141741/99.67	147095/377.67
160	211700/4.33	188504/21.67	189527/14.00	187325/145.00	194321/598.33

Table III(d) : Results for sparse random graphs.

Table III: Average solution values and time required for three runs of different GPP solutions for various graphs. The experiments were conducted on a Pentium 75 running Windows 3.11.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
20	5161/0.00	5064/0.00	5063/0.00	5062/0.40
40	20347/0.20	19992/0.60	20022/0.60	19985/4.30
60	49271/0.40	46822/2.20	45406/1.70	45391/17.30
80	90137/0.40	87105/3.00	81775/4.10	81462/35.80
100	143722/0.80	138856/7.80	126729/6.30	126375/71.00
120	210007/1.20	205801/4.80	186020/11.40	185546/133.00
140	294232/2.00	289504/6.50	255453/16.70	255011/184.80
160	373680/2.50	365529/21.90	323980/32.70	323105/239.40
180	482006/3.20	474430/22.00	415447/41.50	414896/377.50

Table IV(a) : Results for dense Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
20	4331/0.10	4220/0.10	4225/0.00	4193/0.50
40	17555/0.00	17248/0.10	17134/0.30	17047/2.30
60	40125/0.30	39607/0.80	39609/0.80	39386/8.50
80	72410/0.40	71604/2.10	71540/2.20	70917/20.40
100	114549/1.00	113677/3.90	113774/4.40	113032/40.90
120	165021/1.50	163065/6.80	163833/8.10	162811/72.90
140	227353/2.30	224537/10.50	225088/11.40	223882/116.60
160	296995/3.20	292636/19.10	293718/20.00	292610/175.80
180	379129/4.70	371916/30.00	374212/23.60	371825/241.10

Table IV(b) : Results for complete random graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
20	2969/0.00	2768/0.00	2796/0.00	2743/0.10
40	11777/0.20	11376/0.60	11385/0.30	11246/2.20
60	28541/0.50	27669/1.30	27821/1.00	27350/8.10
80	54701/0.90	50173/2.90	49890/2.60	49767/19.10
100	91448/1.10	77350/5.70	77505/4.20	76877/41.30
120	132211/1.80	111338/11.30	111451/7.30	110552/71.10
140	176976/2.50	152744/17.30	152888/11.40	152123/117.10
160	242462/3.30	204851/28.40	204536/19.10	204118/188.00
180	305004/4.40	259067/36.90	259259/26.70	258333/262.50

Table IV(c) : Results for sparse Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
20	2575/0.00	2445/0.00	2458/0.00	2387/0.60
40	10611/0.00	10124/0.40	10161/0.20	10094/2.10
60	24706/0.50	24323/1.00	24631/0.70	24180/7.90
80	46346/0.50	45387/2.10	45546/1.90	44767/20.30
100	73296/1.10	71654/4.00	71361/3.70	70952/41.90
120	108397/1.90	103687/7.30	104483/7.00	103267/71.40
140	155830/3.00	142872/12.70	142863/11.80	141766/113.20
160	209716/4.00	188875/21.40	189244/16.50	187740/171.80
180	263931/5.20	239170/32.90	240517/23.20	238428/246.00

Table IV(d) : Results for sparse random graphs.

Table IV : Average solution values and time required for ten runs of different GPP solutions for various graphs. The experiments were conducted on a Pentium 75 running Windows 3.11.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
100	144067/1.00	141048/3.00	128521/9.50	128518/79.50
140	286673/2.00	281725/6.50	251028/6.50	249190/166.50
180	493570/3.00	487729/12.00	418833/54.50	418059/361.00
220	747892/4.50	741006/21.00	629859/15.50	628081/466.00
260	1045216/7.00	1035385/34.00	878223/52.50	876091/345.00
300	1385825/10.00	1226465/108.50	1167695/40.50	1165996/455.00
340	1755336/12.00	1553463/127.50	1496866/59.50	1493621/677.50
380	2222291/15.50	1873752/137.00	1849515/82.50	1847449/1036.50

Table V(a) : Results for dense Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
100	113903/1.00	113030/3.50	113909/3.00	112247/36.00
140	227313/2.00	224831/11.50	224997/13.00	224444/114.50
180	380484/5.00	373109/25.50	374270/22.00	373048/238.50
220	576581/8.50	561391/48.50	561652/43.50	561022/443.50
260	815302/14.00	792724/104.00	792424/72.00	789287/737.50
300	1091379/21.00	1091379/61.50	1088429/69.50	1081333/738.00
340	1402034/29.50	1402034/117.00	1413690/74.00	1404953/1020.00
380	1752873/40.50	1752873/161.00	1763296/141.50	1761003/1137.50

Table V(b) : Results for complete random graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
100	86917/1.50	76716/6.00	76562/4.00	75864/35.50
140	179700/2.50	156821/17.00	156228/13.00	155882/120.50
180	288753/4.00	253624/36.00	253718/23.50	252400/242.50
220	483358/6.50	396935/65.50	398193/47.50	397167/475.50
260	666977/9.50	565335/124.00	564327/103.00	562773/785.00
300	821884/14.00	781447/93.00	770158/89.00	769344/660.00
340	1065800/18.00	1000549/105.50	983499/74.00	981073/848.00
380	1306380/25.50	1251560/145.00	1250474/101.00	1240290/1071.50

Table V(c) : Results for sparse Euclidean graphs.

Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
100	71977/1.00	70536/3.50	70987/4.50	70171/36.50
140	163316/3.00	146130/12.00	145852/9.00	144371/109.00
180	261490/5.00	237925/27.00	239852/20.50	236517/235.50
220	395293/8.00	362561/51.00	363320/47.00	360996/398.00
260	553320/12.50	514305/102.50	513375/78.50	513356/712.50
300	736056/17.50	727483/86.50	729269/50.00	719488/640.50
340	935415/23.00	935415/82.00	945568/103.00	935767/834.50
380	1174335/31.50	1174335/152.50	1189749/120.50	1178613/1064.50

Table V(d) : Results for sparse random graphs.

Table V : Average solution values and time required for two runs of different GPP solutions for various "large" graphs. The experiments were conducted on a Pentium 75 running Windows 3.11

Graph Type	Nodes	Sol. /Time for GPLA	Sol. /Time for GP_KL	Sol./Time for KLx1	Sol./Time for KLx10
Dense Euclidean	500	3849632/30.00	3259503/628.00	3261159/502.00	3259211/2647.00
Complete Random	500	3052113/86.00	3052113/571.00	3066905/187.00	3057474/2519.00
Sparse Euclidean	500	2171829/51.00	2119500/238.00	2119521/278.00	2111092/2752.00
Sparse Random	500	2040433/57.00	2040433/348.00	2044179/282.00	2037173/2495.00

Table VI: Solution values and time required for a single run of different GPP solutions for a "very large" graph of 500 nodes. The experiments were conducted on a Pentium 75 running Windows 3.11

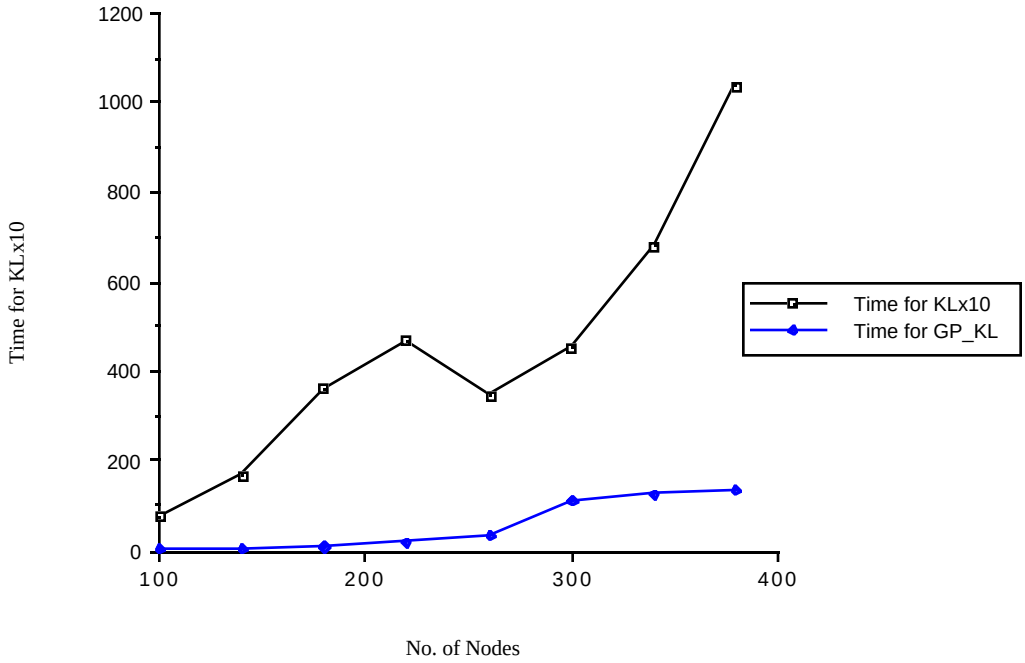

Figure I : The average time required for the GP_KL and KLx10 algorithms for dense Euclidean graphs plotted against the number of nodes. The data for the graph (See Table V(a)) is obtained from experiments conducted on a Pentium 75 running Windows 3.11. Although we have only shown the plot of the times required for these two algorithms, typically, the superiority of our scheme over the current methods seems to increase with the number of nodes.