Using a Genetic Algorithm to Tackle the Processors Configuration Problem

Terry Warwick and Edward P K Tsang

Department of Computer Science
University of Essex
Colchester CO4 3SQ
United Kingdom

Abstract

Distributed programs on irregular networks can outperform those on regular networks. This paper reports on preliminary results in using a Genetic Algorithm (GA) strategy to tackle the Processor Configuration Problem (PCP), the optimisation of irregular multiprocessor network configurations. The PCP is an NP-hard problem and requires the use of heuristic techniques to solve it. Results suggest that the proposed GA provides a useful heuristic strategy to tackling the PCP.

I. Introduction

I.1 Overview

In this paper we try to establish that the Genetic Algorithm (GA) can provide a useful heuristic strategy in tackling the Processor Configuration Problem (PCP), described in detail in section II. This strategy, which we call GAcSP, is a combination of a GA, a repairer and an optional hill-climber. This study is part of a wider research goal of developing a generic Constraint Satisfaction Problem (CSP) solver, introduced in section I.4. The two central issues of this research, namely the GA technique and the CSP are briefly introduced in sections I.2 and I.3. The GAcSP components are described in section III. Section IV presents empirical details, results and discussion on testing the GAcSP. Section V presents empirical results on the use of the hill-climber in the GAcSP strategy. Section VI concludes the paper.

I.2 Constraint Satisfaction Problems (CSP’s)

Many artificial intelligence and computer science problems are instances of CSP’s [7]. These include; scene labeling, graph isomorphism, boolean satifiability, graph colouring, and scheduling (especially resource allocation). The CSP is considered to be NP-complete and requires heuristic techniques to solve it. The efficiency of CSP solving techniques can be improved by using heuristics to guide the process [6]. In this research we use a heuristic approach assisted by local improvement techniques.

The CSP has a finite set of variables, each variable has a finite domain of values and there is a finite set of constraints. A solution tuple is an assignment of a value to each variable (from their respective domains) satisfying the constraints. The Partial Constraint Satisfaction Problem (PCSP) is an optimisation problem, where an objective function can be defined which maps every solution tuple to a numeric value. PCSP’s are CSP’s where there may not be a solution which satisfies the constraints, in which case the requirement is to find the best solution tuple which minimises or maximises the objective function. We suggest that the PCP is an instance of a PCSP and use the heuristic GAcSP approach to tackle it.

I.3 Genetic Algorithms (GA’s)

GA’s are heuristic search techniques for tackling combinatorial optimisation problems. The heuristic is based upon simulating the mechanism of evolution [3]. GA’s will adaptively explore the search space environment by manipulating a set of data structures (i.e. population of chromosomes), and exploit those data structures which are successful. The success of each data structure is measured by the objective function and is usually called its fitness. This strategic balance between exploration and exploitation provides an effective technique for finding near optimal solutions in large search spaces.
A new GA has been designed to tackle PCSP's. Each of the GA components will be discussed in the context of the GAcSP. In the GAcSP the standard GA has been combined with a repairer, and an optional hill-climber, which provides local improvement within the GAcSP.

I.4 Motivation and Objective

The goal of this research is to develop a generic GA as a practical tool for finding optimal or near optimal solutions to PCSP's. Research has already indicated that GA's could provide a useful approach for tackling optimisable constraint satisfaction problems [10]. One instance of the PCSP class of problems is the PCP, which forms the focus of study for this paper. The PCP is an NP-hard problem and presents a significant challenge to the generic GAcSP. The GAcSP strategy used to tackle the PCP has already shown some success on another PCSP, the car sequencing problem (see [8] for more details on this problem). For the GAcSP to tackle the PCP, a domain specific evaluation function was written which would capture the requirements of the problem. The requirements in the PCP are the configuration constraints.

II. The Processors Configuration Problem (PCP)

II.1 Definition

The PCP is the linking together of a finite set of independent processors into a multiprocessor network, where each processor has a fixed number of links available for connection to other processors. The number of links on a processor defines its valency Δ. Motivated by the desire to exploit transputers technology, all PCP's tackled in this paper have processors with valency Δ = 4.

A restriction on the configuration of the network is that two processor links are assumed to be used by the system controller, which acts as input/output to the network. Figure 1 (a) shows a five processor network without input/output. In this research we only consider networks with input/output, such as the network in Figure 1 (b).

The irregular networks that are formed provide useful maps for connecting processors together in distributed memory MIMD machines, and for many applications where performance can exceed that of regular networks [9].

![Figure 1: (a) Network without input/output. (b) Network with input/output.]

II.2 Graph Theory

The multiprocessor network can be described in terms of extremal graph theory [1]: where the processors are the nodes and its links the arcs. Graph theoretic distances can provide useful performance measures for the irregular graphs which are formed. These performance measures
include; diameter, and mean internode distance which can represent limiting factors on communication speed. We can derive a theoretical lower bound for the mean internode distance of optimum graphs, detailed below. The diameter $d_{\text{max}}$ \cite{1} is a graph invariant, and is the maximum direct distance between any two nodes in the graph: where the distance is taken to be the number of nodes which have to be traversed in communication between a source and target node.

The mean internode distance $d_{\text{avg}}$ \cite{1} is the average distance between any two nodes. The mean internode distance for a graph can be calculated as follows:

$$d_{\text{avg}} = \frac{\sum_{p=1}^{N} \sum_{d=1}^{d_{\text{max}}} (d \cdot N_{pd})}{N^2}.$$  \hspace{1cm} (1)

where $N$ is the number of nodes used and $N_{pd}$ \cite{1} is the number of nodes distance $d$ away from node $p$.

A theoretical upper bound on the maximum number of nodes $n_{\text{max}}$ in a graph for a fixed diameter $d_{\text{max}}$ (assuming two arcs are used as input/output), is calculated as follows:

$$n_{\text{max}} = 1 + (\Delta - 1) + (\Delta - 1)^2 + ... + (\Delta - 1)^{d_{\text{max}}}.$$  \hspace{1cm} (2)

The diameter $d_{\text{max}}$ and mean internode distance $d_{\text{avg}}$ measure the compactness of a network. Three benefits of compact networks are:

(a) The more compact the network the shorter the distances for communication, and the less consumption of network links. Compact networks can sustain higher levels of communication.

(b) Compact networks involve shorter internode distances so fewer processors are interrupted from performing useful computations in them.

(c) Communication between two processors involves intermediate processors propagating messages. Compact networks can reduce the latency of communication due to propagation delays.

A lower bound for the mean internode distance, $D_{\text{avg}}$, can be calculated for an optimum graph. An optimum graph is where, each node in an $N$ node graph has the shortest distances possible to all other nodes.

For a node $p$ with $n_{\text{arc}}$ arcs (where $n_{\text{arc}}$ could be 2, 3 or 4), the maximum number of nodes $M_{pd}$ that it can reach in distance $d$ is:

$$M_{pd} = n_{\text{arc}} \cdot 3^{d-1}.$$  \hspace{1cm} (3)

We can calculate the number of nodes $M_{pd_{\text{max}}}$ distance $d_{\text{max}}$ away from $p$ with (see \cite{1}):

$$M_{pd_{\text{max}}} = (N-1) \cdot n_{\text{arc}} \sum_{d=1}^{d_{\text{max}}-1} 3^{d-1}. $$  \hspace{1cm} (4)

The lower bound for the mean internode distance, $D_{\text{avg}}$, for an optimum graph with $N$ nodes, can be obtained from equation:

$$D_{\text{avg}} = \frac{\sum_{p=1}^{N} \sum_{d=1}^{d_{\text{max}}-1} (d \cdot M_{pd}) + M_{pd_{\text{max}}}}{N^2}.$$  \hspace{1cm} (5)
Chalmers and Gregory [1] provide comparable results from their PCP program (AMP), for PCP’s of 32 processors and 40 processors. The maximum number of processors they have configured for a valency $\Delta = 4$, diameter $d_{\text{max}} = 3$ network is 32, whilst the theoretical upper bound $n_{\text{max}} = 40$. The GAcSP approach is to directly optimise the diameter constraint and indirectly optimise the mean internode distance.

Although Prior et al [9] use a GA to tackle the PCP their results are not directly comparable, because we have assumed that two processor links are used by the system controller.

III. Outline of the GAcSP

III.1 Representation

The representation and the GA operators (see figure 2) need to work together in a synergistic way. The GA crossover operator manipulates chromosome-like structures in a way its natural counterpart does. String representations are suitable because of their structural similarity to chromosomes. From the CSP definition given above and its solution tuples we can define a string representation, where each CSP variable relates to a string position and at that string position, has an assigned value taken from the variable domain. For example:

<table>
<thead>
<tr>
<th>String position</th>
<th>CSP Variable</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution tuple</td>
<td>CSP Value</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

From the CSP definition I.2, the set of 10 variables is $\{1, 2, \ldots, 10\}$ and the domain for each variable is $\{1, 2, 3, 4, 5\}$.

In this example, each CSP variable domain (i.e. $\{1, 2, 3, 4, 5\}$) represents the set of processors to be configured by the GAcSP. We can see from the example solution tuple that there are two of each processor, numbered 1 - 5 represented. Each processor in the solution tuple is linked to its immediate neighbours. For example, processor 2 is linked to processors 1, 3, 4, and 5 taking up the four links available. For all strings, the first and last string element have three links available (except when the first and last element represent the same processor), because each uses one link to connect to the system controller. For processor 1 in the above example there are only three links, because as the first element (same for the last string element) it has only one neighbour processor. This representation is compact and ensures that all processor links will be used because each string element will automatically have neighbours. From this description we can see that the representation is limited to PCP’s with even valencies.

III.2 Objective Function

The GAcSP objective function maps each PCP solution tuple to a numeric value. The goal of the GAcSP is to find optimal or near optimal solution tuples or tuple to the PCP. The GAcSP seeks to find a minimum value as defined by the objective function which satisfies the PCP configuration constraint. The constraint in the PCP is the diameter $d_{\text{max}} \geq k$. When the connection in a network configuration has self links or is connected more than once to the same processor, the effect is to increase the string fitness. Self connections and double connections are links which could otherwise reduce a number of the processor to processor paths. The increase in string fitness effectively penalises these strings in favour of strings which do not have self or double connections. The string fitness is calculated by the following equation:

$$\text{ minimise fitness} = \sum_{p}^{N} \sum_{d = (k+1)}^{d_{\text{max}}} N_{pd} \cdot (d - k).$$  (6)
Figure 2: Flow Chart - GAcSP Operator Details

Initialization Operator

- Generate Function
  - Randomly select string value from CSP variable domains, append until string complete.
  - Put string into Population, \( p = POP + 1 \)
  - Evaluate string.

Reproduction Operator

- Elitism Function
  - Select the n fittest individuals (i.e., minimisation - lowest fitness) from the Population.
  - Put strings into MatePool, \( Mp = Mp + n \)

- Select Function
  - Select an individual string from the Population with a proportional bias based upon its fitness and Population average.
  - Put string into MatePool, \( Mp = Mp + 1 \)

Crossover Operator

- UAX Function
  - Randomly select two strings from the MatePool, exchange genetic material to create an offspring.
  - Replace the worst fittest string in MatePool (i.e., minimisation - highest fitness), \( Os = Os + 1 \)

- Greedy Repair Function
  - Repair the offspring by making sure the correct number of each of the CSP values are present in the string; repair choices minimise the string fitness.
  - Evaluate offspring.

Optional Hill-Climbing Function

- Locate high cost string elements, and swap positions with string elements which minimise the string fitness.
- Continue until no further improvement is possible or a pre-set time limit is reached.

Update Variables

- Population = MatePool.
- \( Mp = 0 \)
- \( Os = 0 \)
- \( Cycle = Cycle + 1 \)

End Condition?

- Yes
- Stop

- No

GA cycle

Explanation of Program Variables and Constants

- Cycle = Counter for the number of GA iterations
- \( Mp = \) Counter for the number of MatePool strings generated each Cycle
- \( n = \) GA parameter constant number of fittest members selected each Cycle
- \( Os = \) Counter for the number of offspring generated each Cycle
- OSpring = GA parameter constant for the total number of offspring to be generated each Cycle
- \( p = \) Counter for the number of strings randomly generated during initialisation
- POP = GA parameter constant for the total number of strings in the Population array and MatePool array
where $N$ is the number of nodes used and $N_{pd}$ is the number of nodes distance $d$ away from node $p$.

The fitness is effectively a total measure of all the processor to processor distances, greater than the diameter constraint $k$. As well as the fitness calculated, the mean internode distance is calculated using equation (1) and recorded on the string.

### III.3 Reproduction Operator

The Reproduction operator guides the GA through the search space by selective control using a sampling bias based upon the string fitness. The first stage of the operator implements a technique called "Elitism" [2]. Elitism copies the best fitness strings (i.e. lowest for minimisation) into the mating population (MatePool). This technique guarantees that the "Elite" members of the population will survive into the next generation. (So long as the number of offspring required each generation is less than the (total population - elite copies).) These low fitness strings are considered important because they will have low valued elements or groups of elements (building blocks) in their strings to pass onto their offspring. The second stage of reproduction involves a biased selection from the population, where opportunity for selection is proportional to string fitness and the population average fitness.

### III.4 Crossover Operator

The GA crossover operator explores the structural search space by creating offspring strings from selected parent strings. A crossover operator needs to encourage exploration, yet not destroy the building block information already contained in the population. The crossover operator should enable the offspring to inherit building blocks from the parents.

The GAcSP uses a new crossover operator; the Uniform Adaptive Crossover (UAX), which uses an extended string representation. An example of the extended string representation is:

```
Population member 1- { 1 2 3 4 2 5 4 1 3 5 }
Extended string 1- { 1 2 3 4 2 5 4 1 3 5 } =
                     { 0 1 0 0 1 1 0 0 1 0 }.
```

Each string member of the population will have this extra binary string. So the length of each string member will be doubled to account for this extra information. This extra binary string acts as a template to control the creation of the offspring string during the crossover process. The first stage of the crossover operator is, to randomly select two strings to be used as parents from the MatePool. The UAX operator implemented in the GAcSP constructs a single offspring using the following steps:

**Step.1** - Randomly select one of the two parent strings, call them Parent 1 and Parent 2. Parent 1 is selected to be the starting parent to copy from in Step.2.

```
string position =  1 2 3 4 5 6 7 8 9 10
Parent 1 string
   { 1 2 3 4 2 5 4 1 3 5 }
   { 0 1 0 0 1 1 0 0 1 0 }
Parent 2 string
   { 2 1 3 2 4 1 5 3 4 5 }
   { 1 0 1 0 0 1 1 0 0 1 }
```

**Step.2** - These operations are carried out sequentially, from left to right at each binary position in turn. Examine both parents binary strings at the first binary position: If both parents at the first binary string position have the same binary value, either 0 or 1. Change the parent to copy from the current value to the other parent, (e.g. Parent 1 to Parent 2).

If both parents at the first binary string position have different binary values leave the parent to copy from unchanged.
Example:

\[
\text{string position} = 1
\]

Parent 1 string: 
- \{ 1 \\
- \{ 0 \\

Parent 2 string: 
- \{ 2 \\
- \{ 1 \\

In the example above the parent binary values are different '0' and '1' respectively and therefore the parent used to copy from remains unchanged. Copy the first string element '1' from Parent 1 and copy the first binary element '0' from Parent 1.

Step.3 - Repeat Step.2 for all binary positions resulting in the offspring construction sequence. For example for string position 2, the binary bit of Parent 1 and 2 are '1' and '0' respectively. So the offspring continues to copy from Parent 1. The complete offspring is shown below:

\[
\text{from Parent} \\
\text{Offspring 1 string} = \{ 1 1 1 2 2 1 1 2 2 2 \\
\{ 1 2 3 2 4 5 4 3 4 5 \\
\{ 0 1 0 0 0 1 0 0 0 1 \\
\]

Step.4 - Offspring replaces the worst fitness member of the population.

Step.5 - Repair offspring to ensure it is legal using the greedy repair function below.

Step.6 - Evaluate offspring.

Step.7 - Optional hill-climber.

The technique of using the extended representation is to allow the adaptation of crossover points. The binary string provides a way of recording the successful crossover points and enables the offspring to inherit them. One effect of this new crossover operator upon the representation, is that offspring created will not always satisfy the representation constraints; therefore, each offspring will need to be repaired using a repairer (greedy in this case).

III.5 Greedy Repair Function

Offspring generated from the crossover operator will not always have a legal representation of processors. As we have seen from section III.1, each processor will have two links to each of its two neighbours along the string (except the first and last). Since we are only considering PCPs which have a fixed valency \( \Delta = 4 \); there will need to be two processors in each string to represent the four possible links. The greedy repairer ensures that two of each processor are represented in the offspring string. The repair mechanism is to, first randomly locate a processor in the string which has more than the required two representations. Next, locate all processors which have less than two representations in the string. Change the over-represented processor to the under-represented processor which minimises the string fitness. From this random starting position a sequential search is made for all other over-represented processors, until the representation constraints are satisfied.

III.6 Optional Hill-Climber Function

The GAcSP strategy incorporates an optional Hill-Climber (HC) in order to improve on the quality of solutions. Although the GA is a powerful heuristic technique for finding near optimal solutions to problems, it lacks a local improvement ability. This makes the combination between the GA and HC a synergistic one.

The HC is a simple string element exchange function, for swapping high cost processors with any other processor which will reduce the string fitness. The high cost processors are those with processor to processor paths greater than the diameter constraint constant \( k \). A starting
point is randomly selected, from which high cost processors are located and swapped with processors which minimise the string fitness. This process is carried out until there is no more improvement, or a pre-set time limit is reached.

IV. GAcSP Empirical Work

IV.1 Experiment 1

To allow us to compare our results with other researchers nine separate tests were undertaken on PCP's with \{ 32, 33, \ldots, 40 \} processors. Due to the stochastic nature of the GAcSP there were five runs for each test. All programs are written in C, and tests were run on a SUN 4/110 under UNIX 4.0 operating system. The conditions of all the tests were:

(a) The diameter constraint was set at \( k = 2 \). PCP's with 32 - 40 processors have a minimum diameter of \( d_{\max} = 3 \) [1]. Setting \( k < d_{\max} \) ensures the string fitness is always positive, and makes the hill-climber work harder to improve this fitness, (when switched on).

(b) For each PCP test, the same randomly pre-generated population of 80 strings were used on all 5 runs.

(c) 10\% of the elite population members were copied directly into the MatePool at reproduction phase of the GAcSP.

(d) The number of offspring created each GAcSP cycle was arbitrarily set at four.

(e) The end conditions for each test were set at: maximum number of generations = 300, maximum run-time = 10 CPU hours.

IV.2 Results

All Experiment 1 results have been summarised in Table 1.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower Bound ( D_{\text{avg}} )</td>
<td>2.29</td>
<td>2.31</td>
<td>2.33</td>
<td>2.35</td>
<td>2.37</td>
<td>2.39</td>
<td>2.40</td>
<td>2.42</td>
<td>2.43</td>
</tr>
<tr>
<td>AMP ( d_{\text{avg}} )</td>
<td>2.31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.53</td>
</tr>
</tbody>
</table>

GAcSP Results

| Best \( d_{\text{avg}} \) | 2.33 | 2.37 | 2.40 | 2.42 | 2.42 | 2.45 | 2.47 | 2.51 | 2.51 |
| Best \( d_{\text{eval}} \) | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| paths | 12 | 15 | 24 | 33 | 31 | 47 | 49 | 65 | 62 |
| Avg \( d_{\text{avg}} \) | 2.344 | 2.38 | 2.40 | 2.422 | 2.446 | 2.47 | 2.488 | 2.516 | 2.524 |
| Avg Time Best sec | 1160 | 1101 | 1018 | 1656 | 2095 | 2488 | 5032 | 3240 | 5432 |
| Avg Run-Time sec | 1290 | 1450 | 1701 | 2401 | 2328 | 2967 | 3810 | 4408 | 5143 |

Table 1 explanation.

Lower Bound \( D_{\text{avg}} \) - The theoretical lower mean internode distance bound.
AMP $d_{avg}$ - The best mean internode distances for the AMP program [1].

Best $d_{avg}$ - The best mean internode distance of each of the 5 runs.

Best $d_{eval}$ - Maximum best direct processor to processor distance achieved.

$paths$ - Total number processor paths greater than the minimum diameter for 32 - 40 PCP's, $d_{max} = 3$.

Avg $d_{avg}$ - Average mean internode distance of the 5 runs.

Avg Time Best sec - The average time in CPU seconds taken by the GAcSP to achieve the best mean internode distance.

Avg Run-Time sec - The average of the total times in CPU seconds taken for the GAcSP runs to terminate.

Graph 1: % Mean Internode Distances

IV.3 Discussion

The preliminary results summarised in Table 1 show that, GAcSP can provide near optimal results to PCP's of 32 - 40 processors. However, none of these configurations could satisfy the diameter constraint $k = 2$ or the minimum diameter for 32 - 40 PCP's $d_{max} = 3$. Graph 1 illustrates how close to the lower bound $D_{avg}$ solution can be achieved. The graph also allows us to compare on two points (i.e. 32 & 40 processors) between Chalmers and Gregory [1] PCP program AMP mean internode distance results and that of the GAcSP. Although on the 32 PCP the AMP is 99.1 % of the lower bound whilst that of the GAcSP is 98.3 % lower bound; on the 40 PCP the GAcSP best of 96.8 % is a slight improvement of 96 % for the AMP. It should be noted that the AMP configuration generator is a specially written optimisation program for the PCP [1], whilst the GAcSP is a generic PCSP solver. It should also be noted that there is no evidence to support that the lower bound for the mean internode distance
for PCP's 32 - 40 is achievable.

The total run times and the average CPU times in seconds taken to obtain the best configurations show that near optimal results for 32 - 40 processor PCP's can be obtained in a reasonable period of time. Furthermore, an exponential amount of time is likely to be required by searching in AMP. On the other hand, GAcSP is not too much affected by the scaling problem. Therefore the GAcSP has more potential for solving larger problems.

V. GAcSP with HC Empirical Work

V.1 Experiment 2

We switched on the GAcSP optional HC (see Figure 2) which improves the quality of offspring after being repaired in the crossover operator. The same nine tests were carried out for 5 runs each, with the following conditions:

(a)...(e) As Experiment 1.

(f) The maximum time limit for the HC was 60 CPU seconds.

V.2 Results

All Experiment 2 results have been summarised in Table 2.

Table 2: Summary GAcSP and HC Experiment 2 Results

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower Bound $D_{avg}$</td>
<td>2.29</td>
<td>2.31</td>
<td>2.33</td>
<td>2.35</td>
<td>2.37</td>
<td>2.39</td>
<td>2.40</td>
<td>2.42</td>
<td>2.43</td>
</tr>
<tr>
<td>AMP $d_{avg}$</td>
<td>2.31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.53</td>
</tr>
</tbody>
</table>

GAcSP with HC Results

| Best $d_{avg}$      | 2.29| 2.32| 2.34| 2.37| 2.38| 2.41| 2.43| 2.45| 2.47|
| Best $d_{eval}$     | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   |     |
| $paths$             | 1   | 4   | 3   | 11  | 8   | 18  | 19  | 25  | 34  |
| Avg $d_{avg}$       | 2.344| 2.33| 2.40| 2.422| 2.446| 2.47| 2.488| 2.516| 2.524|
| Avg Time Best sec   | 9103| 4784| 5979| 6046| 10135| 7682| 10304| 11925| 14908|
| Avg Run-Time sec    | 8098| 8784| 8724| 10528| 12050| 11463| 17141| 13314| 18542|

Table 2 explanation.

Lower Bound $D_{avg}$ - The theoretical lower mean internode distance bound.

AMP $d_{avg}$ - The best mean internode distances for the AMP program [1].

Best $d_{avg}$ - The best mean internode distance of each of the 5 runs.

Best $d_{eval}$ - Maximum best direct processor to processor distance achieved.
paths

- Total number processor paths greater than the minimum diameter for 32 - 40 PCP's, $d_{\text{max}} = 3$.

Avg $d_{\text{avg}}$

- Average mean internode distance of the 5 runs.

Avg Time Best sec

- The average time in CPU seconds taken by the GAcSP to achieve the best mean internode distance.

Avg Run-Time sec

- The average of the total times in CPU seconds taken for the GAcSP runs to terminate.

Graph 2: Mean Internode Distances

V.3 Discussion

We can see from the results of Table 1 that the GAcSP can find near optimal solutions but lacks a local improvement ability. We have combined an HC with the GA which gives it a local improvement ability. In Table 2 GAcSP's mean internode distance results are much improved (see graph 2), but at the extra cost in CPU seconds required. These results also show an improvement over the AMP [1] results and are close to the lower bound for the mean internode distance.

VI. Conclusion

In this paper we have outlined a new GA called GAcSP. The GAcSP is a synergistic GA and HC strategy. We have suggested that the PCP is an instance of a partial constraint satisfaction problem and have tackled this problem with the GAcSP. Minimising the diameter constraint and minimising the mean internode distance are the goals of this study. Tests on 32 - 40 processor PCP's have resulted in near optimal solutions within an acceptable time period. The combination of an HC with the GA has improved the quality of solutions but at an extra computational cost. The GAcSP is a general strategy. It has achieved these results with only problem specific data and a domain specific evaluation function which measures the diameter constraint violation. This study forms part of the larger research objective of developing a generic GA PCSP solver. Although the results presented in this paper are only preliminary, they support our work towards establishing a generic GA solver for constraint satisfaction problems.
References


