Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System

Martin Grajcar
grajcar@lrs.uni-passau.de

University of Passau
Chair of Computer Architectures
(Prof. Werner Grass)

Abstract

Our problem consists of a partially ordered set of tasks communicating over a shared bus which are to be mapped to a heterogeneous multiprocessor system. The goal is to minimize the makespan, while satisfying constrains implied by data dependencies and exclusive resource usage.

We present a new efficient heuristic approach based on list scheduling and genetic algorithms, which finds the optimum in few seconds on average even for large examples (up to 96 tasks) taken from [3]. The superiority of our algorithm compared to some other algorithms is demonstrated.

Keywords— heterogeneous system design, heuristic, genetic algorithms, list scheduling

1 Introduction

The solution to our problem consist of two parts:

- Assigning tasks to processors and communications to busses. This part is usually called allocation.
- Determining the start times of all tasks. In some literature the term scheduling denotes this part only.

We try to solve the problem as a whole. For brevity, we call it just “scheduling problem”.

There are many important scheduling problems in the computer science. Their applications range from High Level Synthesis to Hardware-Software Co-design. In the context of Hardware-Software Co-design the scheduler can be used as a subroutine determining, if given partitioning is suitable for given application.

Some extreme simplifications of the scheduling problem can be solved in polynomial time [16, 5], but under realistic assumptions the problem is proven to be NP-hard [8].

We present an heuristic approach to the problem of optimum scheduling for software tasks on a given hardware. Extensions for preemption and for simultaneous optimization of hardware will be also briefly discussed (see section 2.2).

Because of the importance of the scheduling problem, there is a lot of work on it (see section 3).

The computational intractability makes the need for a good heuristic obvious. One important class of scheduling heuristics is list scheduling which considers all tasks in an order given by a list, and selects the best processor for each of them according to some criterion (see section 3.1).

The term genetic algorithms stands for a broad class of algorithms, which exploits an analogy to natural evolution [9, 12]. There is a set of individuals called population, which is initialized at random and maintained according to the “survival of the fittest” principle (see sections 4 and 5).

An individual consists of a chromosome, carrying the genetic information, and some additional data. The best individuals are selected as parents for reproduction. Reproduction is carried out through application of a genetic operator to the parents, which merges their chromosomes, possibly combining good properties of the parents in the child (see sections 6 and 7).

Individuals are rated by interpreting its chromosome, i.e., by building and evaluating a schedule in our case. The chromosome is usually a vector of genes, each of them coding one feature of a candidate solution for the given problem (see section 8).

Experimental results clearly show the superiority of our approach to pure genetic algorithms (see section 9).

2 Problem description

2.1 Basic model

We consider the problem of mapping a task graph onto a target architecture. The task graph is a directed acyclic graph \((T, C)\). The node set \((T)\) contains tasks, and the edge set \((C)\) contains communications. Neither tasks nor communications can be preempted.

The target architecture consists of a set of processing modules \(M\) and a set of buses \(B\). Usually, there is only one bus connecting all modules. Each processing module (module for short) consists of a CPU (responsible for the computation), a local memory, and a communication processor for each connected bus, so computation and communication can overlap. Normally, among the CPU’s there is one general purpose processor, some digital signal processors, and some application specific components (ASIC’s). Additionally, the global memory can be modeled as a processor having zero execution time [2] (so that only the communication times between the memory and other modules are to be considered). Each bus is assumed to be connected to all modules.

For brevity, we define the resource set \(R = M \cup B\) and the user set \(U = T \cup C\). For each user there is a given set of allowed resources \(R_u\) with \(\forall_{u \in U} \, R_u \subseteq M \) and \(\forall_{c \in C} \, R_c \subseteq B\). The assignment set is defined as \(A = \{(u, r) \in U \times R \mid r \in R_u\}\). The execution time \(d_{ur} = [0, \infty)\) of a user \(u\) on a resource \(r\) is assumed to be given for each \((u, r) \in A\).

The result of our algorithm is a schedule defined as a pair \([\text{ass}, \sigma]\) with
The model described above is quite simple, because we wanted to concentrate on the algorithm development, rather than to generalize an existing algorithm.

However, there is one important model extension handled “for free” by the algorithm developed: the preemption. User $u$ can be preempted by another user $u'$, i.e., $u$ suspends its execution, and after $u'$ completes, $u$ is continued on the same resource. Of course, there is some time penalty for saving and restoring the context (i.e., register contents and so on) of $u$. A task may be preempted more times, and preemptions may be nested.

In spite of the time penalty, preemption can be extremely useful, e.g., if there are some long-running tasks, as figure 1 shows.

![Figure 1: There are two identical modules. Execution times are given in the task graph. Preemption overhead is assumed to be 2 time units and communication times are set to zero for simplicity.](image)

In some models continuation of a user on a different resource (so-called migration) is also considered. Such models assumes a homogeneous processor system, because (as far as we know) there is no operating system for a heterogeneous processor system supporting task migration. For this reason, we do not handle migration.

### 2.2 Model extensions

#### 2.2.1 Preemption

The model described above is quite simple, because we wanted to concentrate on the algorithm development, rather than to generalize an existing algorithm.

However, there is one important model extension handled “for free” by the algorithm developed: the preemption. User $u$ can be preempted by another user $u'$, i.e., $u$ suspends its execution, and after $u'$ completes, $u$ is continued on the same resource. Of course, there is some time penalty for saving and restoring the context (i.e., register contents and so on) of $u$. A task may be preempted more times, and preemptions may be nested.

In spite of the time penalty, preemption can be extremely useful, e.g., if there are some long-running tasks, as figure 1 shows.

- $\text{ass}: U \rightarrow R$ determining the assignment of each user to a resource (to be executed on)
- $\sigma: U \rightarrow [0, \infty]$ determining the start time of each user

satisfying the conditions stated below.

We extend the definition domain of $d$ to the whole set $U \times R$ as follows. Communication is allowed to be executed internally on a module $m$, if both its source and destination are assigned to $m$, i.e.,

$$\forall u \in U \quad \forall (t, r) \in C \quad \text{ass}_{u(t,r)} = m \Rightarrow \text{ass}_r = \text{ass}_u = m$$

We assume that such a communication needs no time, i.e., $\forall (t, r) \in C \\lor (r, m) = 0$. The values $d_{u,r}$ of all remaining pairs $(u, r)$ correspond to impossible assignments and are therefore set equal to infinity.

Each user $u$ uses its resource $\text{ass}_u$ in the whole execution interval $[\sigma_u, \eta_u]$ exclusively, where

$$\forall u \in U \quad \eta_u = \sigma_u + d_{u, \text{ass}_u}$$

denotes the end time of $u$. In other words, two different users $u$ and $u'$ use different resources, or their execution intervals do not overlap, i.e.,

$$\forall u, u' \in U \quad \text{ass}_u \neq \text{ass}_{u'} \lor \{ \sigma_u, \eta_u \} \cap \{ \sigma_{u'}, \eta_{u'} \} = \emptyset$$

Using the task graph $(T, C)$ the predecessor set $\text{pred}_u$ of a user $u$ is defined as follows: The predecessor set of communication $c = \{t, r\}$ consists of only the task $t$ i.e., $\forall (t', r) \in C \quad \text{pred}_{(t', r)} = \{t\}$. The predecessor set of a task $t$ is the set of all incoming communications, i.e., $\forall t \in T \quad \text{pred}_t = \{ (t', r) \in C : r = t \}$.

A user can start execution only after having received all its data, i.e., after all its predecessors have finished (AND-causality). Of course, there is at least one task having no predecessor. Each such task can start at time 0.

It is useful to formulate this condition using the ready time $\rho_u$ of a user $u$, which computes as follows:

$$\forall u \in U \quad \rho_u = \begin{cases} 0 & : \text{pred}_u = \emptyset \\ \max \{ \rho_v \in \text{pred}(u) \} \eta_v & : \text{pred}_u \neq \emptyset \end{cases}$$

Now, the condition can be stated as

$$\forall u \in U \quad \sigma_u \geq \rho_u$$

The goal is to find the schedule with the least makespan defined as $\mu = \max_{u \in U} \eta_u$, which satisfies the conditions described above.

### 3 Previous related research

The multiprocessor scheduling problem has been studied very extensively [1, 2, 3, 6, 10, 11, 14]. Unfortunately, there is a lot of variants of the scheduling problem, so direct comparison of two works is hardly possible.

For example, many algorithms assume a homogeneous processor system, and/or ignores communication overhead (e.g., [10] combines critical path and branch-and-bound algorithms, but do not consider communication overhead).

Some algorithms consider communication times, but without any precedence relations [7], so the problem reduces to a kind of clustering problem. We feel that two communicating tasks have to be executed in a given order, and therefore the precedence relations are to be considered.

Other algorithms try to find the shortest schedule assuming an infinite number of identical processors (e.g., the Dominant Sequence Clustering Algorithm as described in [11]).

For solving the scheduling problem various approaches has been taken, such as list scheduling (see section 3.1), genetic algorithms [6], tabu search [13], or linear programming [14, 2, 3].

The problem considered in this paper is very similar to the one from [2], but the solution approach differs. In [2] an optimum
scheduler is described based on linear programming. This scheduler was used to obtain optimum schedules for the examples presented in section 9. Obviously, given enough time, an optimum scheduler is better than a heuristic, but for large examples the computational effort is prohibitive.

### 3.1 List Scheduling Algorithms

The list scheduling algorithms described in the literature use mainly the terms “task” and “module”, because they deal with communications and busses indirectly or not at all. We use the more general terms “user” and “resource”, so this formulation can be used for the description of our algorithm.

**List scheduling** is a large class of algorithms, which considers tasks one by one, until a valid schedule for all tasks is obtained. In our scheme, the list scheduler employs the following sets:

- \( R \subseteq T \) the remaining set consisting of users yet to be scheduled
- \( E \subseteq R \) the eligible set consisting of users, which can be scheduled in the current step
- \( F_r \subseteq [0, \infty) \) the free set of a resource \( r \) being a union of intervals, at which \( r \) is not in use

These sets are initialized by setting \( R := T, E := \{ u \in R : \text{pred}_u = \emptyset \}, \) and \( F_r := [0, \infty) \). Thereafter the following three steps are repeated until \( R \) is empty:

- **User selection** Out of the eligible set, one task \( t \in E \) is selected.
  
  Usually, the set \( E \) is implemented as a list (that is where the term “list scheduling” comes from) sorted in decreasing order of user priorities \( \pi_u \).

- **Resource selection** A resource \( r \) is selected for user \( u \) according to some criterion. It is convenient to assign at this step priorities \( \pi_{ur} \) to resources, and to select the resource with the highest priority. For example, the priority \( \pi_{ur} \) can be set equal to the negative of the earliest possible start time of user \( u \) on resource \( r \), so the resource minimizing the start time of \( u \) will be selected as in [6],

- **Updating of the sets** User \( r \) is removed from the remaining set and from the eligible set, the eligible set is extended by all users having no more predecessors in the remaining set, and the free set of resource \( r \) is updated by removing the execution interval of user \( u \). Formally, the following assignments are made

  \[
  R := R \setminus \{ u \} \\
  E := \{ u \in R : \text{pred}_u \subseteq U \setminus R \} \\
  F_r := F_r \setminus [\sigma_u, \eta_u)
  \]

Later considered user can execute earlier, if there is an appropriate interval in \( F_r \) (this feature is called “hole filling”). Of course, this way better schedules can be obtained, because the requirements on the user priority assignment are relaxed a little.

For efficiency, the free set should be implemented as a tree of intervals. For simplicity, sometimes only the last of this intervals is maintained (which can be represented solely by its starting point). This can be expressed by the assignment

\[
F_r := F_r \setminus [0, \eta_u)
\]

Usually, the user priorities are recomputed on every iteration (dynamically).

Obviously, not every list scheduling algorithm fits in the above scheme, e.g., the Edge Zeroing algorithm [15] does not. Because of limited space, we can describe only few scheduling algorithms here, and we have to simplify radically.

The Modified Critical Path algorithm [17] assigns priority to user \( u \) equal to the length of the longest path starting in \( u \) (the path length for the successor nodes is used to break ties). So users likely to lie on a critical path get high priorities.

Among all list scheduling algorithms the Dynamic Critical Path Scheduling seems to be the sure winner [11]. It is too complicated to be described here, so we just briefly outline some features of it:

- It assigns priorities dynamically based on the dynamic critical path (defined as the most time consuming user sequence on the current scheduling step).
- The start times are not fixed until all users have been considered.
- It selects a suitable resource for a user by looking ahead at the potential start time of the so called critical child of the user.

Further list scheduling algorithms can be found in [1, 11].

### 3.2 Genetic List Scheduling Algorithms

Approaches combining list scheduling and genetic algorithms are taken in [6] and in this paper. Instead of computing priorities once using a sophisticated algorithm, the priorities are computed many times, and the corresponding schedules are build and evaluated.

At the beginning, a couple of priority vectors is generated and stored in chromosome of individuals forming the initial population. These vectors can be generated randomly, or by some heuristic. The population is maintained using a genetic algorithm (see sections 4 to 7), with the fitness determined using a list scheduler (see section 8).

Every chromosome corresponds to a schedule, so many different schedules can be tried, and the best of them is returned upon termination. Any suitable termination criterion can be used, e.g., the genetic algorithm finishes after having found an acceptable schedule, or after having consumed all available time.

The algorithm described in [6] suffers from employing a poor list scheduler (our algorithm performs much better, see section 9).

### 4 Main loop

Our algorithm is a kind of genetic algorithm using list scheduling. There are too many different kinds of genetic algorithms to be discussed here, so we briefly describe our way and outline the most important features.

The whole algorithm repeats in a single loop. After initializing the population with random individuals, the following steps are repeated until some termination criterion is met:

- Select individuals (parents) for reproduction and an individual (looser) to be removed (see section 5)
- Select a genetic operator (see section 6)
- Create a new individual (child) by applying the selected operator on the selected individuals (see section 7)
• Evaluate the new individual using list scheduling (see section 8)
• Replace the looser by the child and make some bookkeeping

Replacing the individuals one-by-one (rather than replacing whole generations), makes the algorithm less sensitive to population size (this method is called steady-state genetic algorithm [12]). We remember the best schedule found so far, rather than preventing the replacing of the best individual (elitist model, see [12]).

5 Individual selection

The evaluation of an individual is done by creating a schedule using its chromosome (see section 8) and computing the best chromosome. The makespan is to be minimized, and better individual should have higher fitness value, so we set fitness = −μ.

We don’t use fitness based selection because of problems mentioned in the literature:

• decreasing selection intensity due to converging towards the optimum [4]
• premature convergence problem due to creation of super-individuals [12]

Instead, we compute the rank of an individual i as the number of worse individuals in the population P, i.e.,

\[
\text{rank}(i) = |\{j \in P : \text{fitness}(j) < \text{fitness}(i)\}|
\]

We compute the chance of an individual i according to

\[
\text{chance}(i) = \frac{\text{rank}(i)}{\text{nc}(i) + 1}
\]

where \text{nc}(i) the number of children of the individual i denotes.

The parents are selected out of some random population subset as the individuals having the greatest chance. From another random population subset, the individual having the least chance will be replaced by a new individual.

Using random population subsets (instead of the whole population) makes the algorithm probabilistic (a genetic algorithm is supposed to be probabilistic) and reduces the computation overhead.

Using the number of children in the computation prevents the best individuals from getting too much children, thus protecting the population against premature convergence.

For population size \(|P| = 100\) subsets with about 10 elements seems to be the best choice.

6 Operator selection

As there are multiple genetic operators in our algorithm, we need to select one of them on each step of the main loop. We define the fitness of an operator as

\[
\text{fitness}(o) = \frac{|P| + \sum_{\text{children}(o)} \text{rank}(i)}{|\text{children}(o)| + 1}
\]

where \text{children}(o) denotes the set of individuals produced by operator o. An operator is selected with a probability proportional to its fitness (so called “roulette wheel selection”).

7 Genetic operators

We achieve efficiency by sophisticated interpretation of chromosomes (see section 8), instead of relying on a complicated problem-dependent representation. The chromosomes used are just vectors of genes (i.e., real numbers), so genetic operators can be kept simple.

We define a genetic operator as a function taking any fixed number of arguments (i.e., chromosomes) and returning one chromosome. This definition is quite general, as it includes

• Operators taking no arguments, and generating a new chromosome as a random vector. Such operators are used mainly for initializing the population.
• Operators taking one argument, which corresponds to so called mutation.
• Operators taking two arguments, which corresponds to so called crossover, except for crossover returning two chromosomes. To our knowledge, the only reason for producing two chromosomes at once, is the analogy to the natural crossover.
• Operators taking more than two arguments (see below).

Because of space limitations, we describe just one (quite uncommon) implemented genetic operator, that proved to be quite successful in producing good chromosomes, while at the same time increasing population diversity. This operator takes three arguments, and computes the result gene by gene, setting every gene with a probability of 0.5 equal to the minimum of the corresponds genes in the parents, else setting it equal to its maximum. This way the operator increases the frequency of extremal values in the population, what leads to increased population diversity.

8 Schedule generation

In previous section the chromosome was defined to be just a vector of genes (real numbers), but now the meaning of elements of genes is to be explained. We employ \(|U| + |A|\) genes: a chromosome contains one gene \(y_u \in \mathbb{R}\) for each user \(u \in U\) and one gene \(y_{u,r} \in \mathbb{R}\) for each assignment \((u, r) \in A\).

These genes are used in computing priorities for the list scheduler, which, on principle, works as described in section 3.1, with the only difference being the re-considering of communications described below.

How priorities are computed is yet to be explained. This computation (described below) is easier to understand using the following observations made during design of the algorithm.

• Communications compete for busses the same way as tasks compete for modules. So it is useful to assign priorities to all users.
• A communication \((t, t')\) is allowed to be executed internally on module \(m\), iff both \(t\) and \(t'\) are assigned to \(m\). The list scheduling algorithm as described in section 3.1 considers a user only after having considered all its predecessors. It means, that while considering the communication \((t, t')\) the assignment of the task \(t'\) is not yet known, so the communication can not be assigned to any module.

The simplest solution to this problem is to consider after having scheduled a task all its incoming communications once again, possibly assigning them to a module, so the communication delay can be eliminated. More complicated solutions like forcing the internal execution using a special gene were also tried, but none of them works as good as the simplest solution.
This is the place, at which tasks and communication have to be treated differently.

- Our genetic algorithm finds critical users quickly and assigns them high priorities, so good schedules are available soon.
- It is very useful to assign a time critical user \( u \) to a resource, on which it can finish as early as possible (given its ready time \( p_u \) and the free sets \( F_r \) as defined in sections 2.1 and 3.1, respectively). Assigning a user to a resource on which it can start early does not help at all if the corresponding execution time is too long.
- Sometimes a task \( r \) has to be assigned to a module on which it does not finish as early as possible. One possible reason can be a costly communication to another task \( r' \), which is better to be executed on another module (see figure 2).

Because of this problem, the genes \( y_{ur} \) were introduced.

We interpret the corresponding genes directly as user priorities, i.e., we set \( \pi u := y_{ur} \). As already stated, it works well, as critical tasks get quickly assigned high priorities.

In order to assign a user to a resource on which it can be finished as soon as possible, we could set the corresponding priority \( \tau u,r := -\tau u,r \) with

\[
\tau u,r = \min \{ \eta u \in \mathbb{R} : [\alpha u, \eta u] \subseteq F_r / \sigma u \geq p_u / \eta u = \sigma u + d_{u,r} \}
\]

According to this equation \( \tau u,r \) denotes the time at which user \( u \) finishes, provided it is assigned to resource \( r \) and executes after its ready time \( p_u \) in the first interval contained in the free set \( F_r \), which can accommodate its execution interval.

In order to assign a user to a resource determined solely by a chromosome, we could simply set \( \pi u,r := y_{u,r} \). Actually, we wish to combine the above expressions, but, in general, they are not of a comparable magnitude.

The value of \( y_{u,r} \) lies (because of the way the initial population is created) usually in the interval \([-1, 1]\), and \( \tau u,r \) lies in the interval \([0, \hat{\mu}]\), where \( \hat{\mu} \) is an upper bound on \( \mu \). With the setting \( \pi u,r := y_{u,r} - \frac{\tau u,r}{\hat{\mu}} \) both expressions can be decisive.

We generalized the algorithm to handle preemption (even multiple or nested) with only a negligible overhead. The only difference is the use of a modified formula for computing \( \tau u,r \), where instead of searching for an interval \([\pi u, \eta u] \) of a length \( d_{u,r} \) fitting in the free set \( F_r \), \( n \) intervals are searched for, such that their total length is enough for the execution delay plus \( (n - 1) \) times the preemption penalty. Since the algorithms used for comparison in the next section do not handle preemption, and because of the limited place, we do not describe the details here.

### 9 Experimental results

Our genetic list scheduling algorithm was evaluated using examples taken from literature (see below). Our algorithm performs for each of these examples better than the other, but this comparison is not especially fair, because of the following differences:

- The algorithms described in [14] and [2] are optimum schedulers, i.e., they found the optimum and prove that there is no better solution. The algorithm from [6] and our algorithm are heuristics (so they may return good results, but they can not prove them to be close to the optimum).
- On the other hand, our algorithm was run until finding a solution at least as good, as the one found by the compared algorithm. Often our algorithm took only a small fraction of the time required by the other algorithms.
- The other algorithms allow for optimizing the hardware cost, e.g., by saving a module.
- Our algorithm cannot optimize hardware yet (but see section 2.2.2), but it can do other things, such as scheduling preemptible tasks.

The first set of examples is taken from [6]. The columns of the table 1 are explained here:

- **9a** This is the nine-task example from [14] and [6]. All available resources are used.
- **9b** For the algorithms from [14] and [6], the run time is measured for the example 9a using different settings (so speed is traded for hardware savings). For our algorithm, this is the example 9a after having removed all resources not used by the solution given in [6].
- **25a** This is the example from [6], which is too large to be solved by the method described in [14], so the optimum is unknown. According to the cost, all resources are used.
- **25b** This is the same example, except for the requirement, that the makespan must be one time unit shorter.
- **25c** This is once again the same example as 25a, but with the requirement, that the makespan must be two units shorter.

As the only result reported in [6] for the 25 tasks example is the one obtained after 9 seconds (\( \mu = 23 \)), we suppose, that no better result was found. This can be explained by examining the list scheduling algorithm employed: A task is assigned to the module on which it can start as early as possible. In a heterogeneous processor system it means, that a task can be assigned to a module on which its execution time is very large, so this task finishes very late. This actually happens for the 25 tasks example.

The second set of examples was obtained from the author of [2]. For each of them the number of modules, busses, tasks and communications is given.

---

1These examples are available from the author of this paper via email (gra@csirs.unipr.sns.snc.de).
Table 2: Comparison for the second set of examples (described in text). All times were obtained using a SPARC 20 workstation. Our results are average values obtained from 100 runs (terminated after the optimum was found).  

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Run Time</th>
<th>Our Run</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[2]</td>
<td>Time</td>
</tr>
<tr>
<td>[M] [B] [T]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 2 12 16</td>
<td>0.7s</td>
<td>0.01s</td>
</tr>
<tr>
<td>4 2 10 13</td>
<td>39.3s</td>
<td>0.06s</td>
</tr>
<tr>
<td>2 1 8 13</td>
<td>2.1s</td>
<td>2.58s</td>
</tr>
<tr>
<td>3 0 26 16</td>
<td>8.15m</td>
<td>0.06s</td>
</tr>
<tr>
<td>8 1 18 31</td>
<td>8.3s</td>
<td>0.06s</td>
</tr>
<tr>
<td>8 1 26 48</td>
<td>2.4m</td>
<td>0.15s</td>
</tr>
<tr>
<td>8 1 30 50</td>
<td>1.2m</td>
<td>0.42s</td>
</tr>
<tr>
<td>8 1 32 57</td>
<td>2.5m</td>
<td>0.10s</td>
</tr>
<tr>
<td>8 1 52 98</td>
<td>52.4m</td>
<td>0.62s</td>
</tr>
<tr>
<td>8 1 64 121</td>
<td>39.4m</td>
<td>1.01s</td>
</tr>
<tr>
<td>8 1 80 2</td>
<td>59.1m</td>
<td>7.88s</td>
</tr>
<tr>
<td>8 1 96 183</td>
<td>3.06h</td>
<td>4.77s</td>
</tr>
</tbody>
</table>

10 Conclusion

The combination of genetic and list scheduling algorithms proved to be very powerful, at least in solving scheduling problems found in the literature. Our algorithm is not as general as the algorithms it was compared to, but it clearly outperformed them for the restricted problem.

Being a heuristic, our algorithm cannot be used to prove a given solution is optimal. Nevertheless, it was able to find the optimum solution for all problems solved to optimum by the other algorithms.

To our opinion, it is possible to generalize our algorithm without any substantial performance loss. This generalization and the investigation of its behavior while solving larger problems are topics of further research.

11 References