# THE SWAPPING HEURISTIC

By

Brian Zulawinski

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1995

## ABSTRACT

THE SWAPPING HEURISTIC

By

Brian Zulawinski

In a grouping problem, a set must be partitioned subject to problem-specific constraints. The Swapping Heuristic (SH) is a family of local search heuristics for solving grouping problems. In this paper we apply the SH to two classic grouping problems: the Bin Packing Problem and the Minimum Makespan Problem. We show that the SH outperforms previously published methods for solving these two problems in a wide variety of situations.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# 1. Introduction

In a grouping problem, a set must be partitioned subject to problem-specific constraints. The Swapping Heuristic (SH) is a family of local search heuristics for solving grouping problems. In this paper we discuss the SH as applied to two classic grouping problems: the Bin Packing Problem and the Minimum Makespan Problem.

Research on these problems falls into two distinct categories: theoretical and practical. Theoretical research focuses on determining the computational complexity and proving bounds on the worst-case performance. These analyses are usually limited to simple deterministic algorithms and approximations schemes. The simple algorithms do not produce high quality approximations. The approximation schemes are difficult to implement and require unreasonable execution times for solutions with small error. Practical research focuses on search tools including simulated annealing, genetic algorithms, heuristics, and various combinations of these methods. These methods are difficult to analyze theoretically but produce good results with practical execution times. These methods are often complex and involve "tweaking" many parameters. The SH is a practical approximation scheme that produces high quality results in short amounts of time. The SH is also easier to implement than other complex solutions found in the literature.

# 2. The Bin Packing Problem and the Minimum Makespan Problem

Figure 1 defines the Bin Packing Problem (BPP), a well-studied grouping problem. Figure 2 describes the Minimum Makespan Problem (MMP), the dual of the Bin Packing Problem. In the Bin Packing Problem, the size of the bins is fixed and the number of bins is minimized. In the Minimum Makespan Problem, the number of machines (bins) is fixed and the makespan (size of the bins) is minimized.

Given:
- Identical bins of capacity $C$
- A set $W = \{1,2,\ldots,n\}$ of $n$ objects with sizes $s_i$, $i \in W$ such that each $s_i \le C$

Constraints:
- Each object in $W$ is assigned to exactly one bin.
- The sum of the object sizes of the objects assigned to any bin does not exceed the bin capacity $C$.

Goal:
- Find an assignment of the objects to bins such that the number of bins, $m$, is minimized.

**Figure 1: The Bin Packing Problem**

Given:
- a set of $n$ jobs with designated integral processing times $p_j$
- $m$ identical machines.

A *schedule* of jobs is an assignment of the jobs to the machines, so that each machine is scheduled for a certain total time, $t_i$. The maximum time that any machine is scheduled for is called the *makespan* of the schedule. Therefore, $Makespan = \max_{i=1,m}\{t_i\}$.

Goal:
- Find a schedule that minimizes the makespan.

**Figure 2: The Minimum Makespan Problem**

## 3. Theoretical Background and Limitations

The time requirements of an algorithm are expressed in terms of the "size" of a problem instance [Garey and Johnson, 79]. The relative difficulty of problem instances varies with several factors, including their size. The size of an instance of the BPP is the number of objects. The size of an instance of the MMP is the number of jobs. The time complexity function for an algorithm expresses its time requirements by giving, for each possible input size, the largest amount of time needed by the algorithm to solve a problem instance of that size. A *polynomial time algorithm* has a polynomial time complexity function. An algorithm that cannot be bounded by a polynomial time complexity function is a *superpolynomial time algorithm.* The computational time required for superpolynomial time algorithms grows far more rapidly than polynomial time algorithms with respect to the input size. Therefore, superpolynomial time algorithms are impractical for large problem instances.

To date there is no known polynomial time algorithm for solving the BPP or the MMP. Furthermore, these problems belong to the class of NP-Complete problems. No member of the class of NP-Complete problems has a known polynomial time algorithm. In addition, if a polynomial time algorithm is found for any member of the class of NP-Complete, every problem in NP-Complete would also be solved in polynomial time. Currently, all algorithms which produce optimal solutions for any NP-complete problem are superpolynomial time algorithms.

Since superpolynomial time algorithms are impractical for large problem instances, the focus turns to finding near-optimal solutions that require less time. There are simple, practical algorithms which achieve small constant approximation ratios. However, we often require optimal or extremely close to optimal solutions. Another approach is polynomial approximation schemes. For any bounded error, there is a polynomial time algorithm which can guarantee solutions within that error bound. These schemes have a high-order polynomial complexity for small error and are often not feasible for practical problems. These algorithms are also complex and difficult to implement. Thus, practitioners have often turned to other methods which do not have provable worst-case guarantees, which do not have provable running times, but which empirically produce good results quickly. Our work presents a newer algorithm along these lines which is simpler, often runs faster, and often produces better solutions than these previously used practical algorithms.

# 4. Known Methods for the Bin Packing Problem

## *4.1 Theoretical Results*

### 4.1.1 First Fit

First Fit is arguably the simplest approach for the BPP.

**First Fit (FF):** For each object, place the object in the first bin it fits. If no such bin exists, start a new bin and place the object in the new bin.

FF has a time complexity function $O(n^2)$. For any instance $I$, the number of bins used by FF is upper bounded by Equation 1 [Garey and Johnson, 79].

**Equation 1**

$$FF(I) \le \left(\frac{17}{10}\right) Opt(I) + 2$$

$FF(I)$: Number of bins used by First Fit on Instance $I$

$Opt(I)$: Minimum number of bins required for Instance $I$

### 4.1.2 First Fit Descending

First Fit Descending improves on FF by sorting the objects in descending order before they are assigned to the bins.

**First Fit Descending (FFD):** Sort the objects in descending order. For each object, place the object in the first bin into which it fits. If no such bin exists, start a new bin and place the object in the new bin.

FFD also has a time complexity function $O(n^2)$. For any Instance $I$, the number of bins used by FFD is upper bounded by Equation 2 [Garey and Johnson, 79].

**Equation 2**

$$FFD(I) \le \frac{11}{9} Opt(I) + 4$$

$FFD(I)$: Number of bins used by First Fit Descending on Instance $I$

$Opt(I)$: Minimum number of bins required for Instance $I$

### 4.1.3 Best Fit

Best Fit tries to improve on FF by placing each object into the bin with the minimum available space that can still accommodate the object. The bin meeting this criterion is said to have the "best" fit for the object.

**Best Fit (BF):** For each object, place the object in the bin that has current contents closest to, but not exceeding $C - s_i$. If no such bin exists, start a new bin and place the object in the new bin.

BF has a time complexity function $O(n^2)$. BF's worst-case performance is identical to FF.

### 4.1.4 Best Fit Descending

Best Fit Descending improves on BF by sorting the objects before they are assigned to the bins.

**Best Fit Descending (BFD):** Sort the objects in descending order. For each object, place the object in the bin that has current contents closest to, but not exceeding $C - s_i$. If no such bin exists, start a new bin and place the object in the new bin.

BFD also has a time complexity $O(n^2)$. Surprisingly, BFD's worst case performance is identical to FFD.

## 4.2 Approximation Schemes

For any relative bounded error $\varepsilon$, polynomial approximation schemes provide an algorithm with a polynomial time complexity function. The time complexity functions for small $\varepsilon$ are high order polynomials, making these schemes computationally infeasible for practical problems.

### 4.2.1 Asymptotic Polynomial Approximation Scheme

An asymptotic polynomial approximation scheme (APAS) is a family of algorithms $\{A_\varepsilon | \varepsilon > 0\}$ such that each $A_\varepsilon$ runs in polynomial time with respect to the length of the input and $A_\varepsilon(I) \le (1+\varepsilon)\cdot Opt(I)$ as the length of the input goes to infinity. [Vega and Lueker, 81] identify an APAS which runs in linear time but is severely exponential in $\varepsilon$. Their method produces solutions that are upper bounded by Equation 3.

**Equation 3**

$A_\varepsilon(I) \le (1+\varepsilon)\cdot Opt(I) + 1$
$A_\varepsilon(I) \le (1+\varepsilon)$ as $n$ goes to infinity
$n$: Number of objects
$A_\varepsilon(I)$: Number of bins used by $A_\varepsilon$ on instance $I$
$Opt(I)$: Minimum number of bins required for Instance $I$

### 4.2.2 Asymptotic Fully Polynomial Approximation Scheme

An asymptotic fully polynomial approximation scheme (AFPAS) is a family of algorithms $\{A_\varepsilon | \varepsilon > 0\}$ such that each $A_\varepsilon$ runs in polynomial time relative to the length of the input and $1/\varepsilon$, while $A_\varepsilon(I) \le (1+\varepsilon)\cdot Opt(I)$ as the length of the input goes to infinity. [Karmakar and Karp, 82] propose a AFPAS with running time $O(\frac{n\log n}{\varepsilon^8})$. The algorithms produce solutions that obey Equation 4.

**Equation 4**

$$A_\varepsilon(I) \le (1+\varepsilon)\cdot Opt(I) + \frac{1}{\varepsilon^2} + 3$$

$A_\varepsilon(I) \le (1+\varepsilon)\cdot Opt(I)$ as $n$ goes to infinity

$n$: Number of objects

$A_\varepsilon(I)$: Number of bins used by $A_\varepsilon$ on instance $I$

$Opt(I)$: Minimum number of bins required for Instance $I$

### 4.2.3 Near-Absolute Approximation Scheme

The Near-Absolute Approximation Scheme is a modification of the work of Karmakar and Karp [Johnson, 82]. This AFPAS has absolute error bounded by a polylogarithmic function of the optimal solution. The solutions obtained with this method are upper bounded by Equation 5.

**Equation 5**

$$NAAS(I) \le Opt(I) + O(\log^2 Opt(I))$$

$NAAS(I)$: Number of bins used by the Near - Absolute Approximation Scheme for instance $I$

$Opt(I)$: Minimum number of bins required for instance $I$

## *4.3 Practical Results*

### 4.3.1 Open Loop Hybrid Algorithm

[Bilchev, 94] proposes an Open Loop Hybrid Algorithm (OLHA) consisting of a Genetic Algorithm (GA) and a Multi-agent System (MAS). In order to describe the system, the MAS and the GA are described first.

The MAS performs exchanges of objects between two bins that increase the value of the objective function given by Equation 6.

**Equation 6**

$$f_{MAS} = A \cdot \Delta_{\text{space}} + B \cdot \Delta_{\text{num of objects}}$$

$A$: Positive Constant

$B$: Positive Constant

$\Delta_{\text{space}} = (\text{Size of more full bin})_{\text{final}} - (\text{Size of more full bin})_{\text{initial}}$

$\Delta_{\text{num of objects}} = (\text{Num of objects in less full bin})_{\text{final}} - (\text{Num of objects in less full bin})_{\text{initial}}$

A genetic algorithm is a search algorithm based on natural selection and natural genetics [Goldberg, 89]. GAs are more robust than conventional search methods. GAs can be applied to a wide array of problems and produce near-optimal solutions on difficult search-spaces.

GAs require the problem's parameter set to be coded as a finite-length string over some finite alphabet. The strings are commonly called *chromosomes*, their analogue in biology. Each chromosome represents a complete solution to the problem. The mapping of the parameters onto the chromosome is called the *representation*. GAs directly manipulate the chromosomes, not the parameters themselves.

Each chromosome has a corresponding *fitness*, a scalar measure of the quality of the chromosome. GAs make use of an *objective function*, a function that evaluates the fitness of a chromosome. GAs sample the search-space using only fitness values of the samples to guide the search. GAs are said to be *blind* since they require no knowledge other than fitness values. Since every problem has some metric for evaluating the quality of a solution, GAs can be applied to many problems.

GAs search using a set of chromosomes called a *population*. A population of chromosomes enables a GA to search many regions of the search-space simultaneously, a desirable characteristic for searching multi-modal search-spaces. A set of chromosomes contains more information than the chromosomes themselves. Similarities among the chromosomes and their relationship to fitness values can be analyzed. Important similarities among highly fit chromosomes are called *schema* or *building blocks*. Building blocks are small parts of a complete solution. The GAs *implicitly* analyze and process schema by using a survival of the fittest and a structured yet randomized information exchange between chromosomes.

GAs start with an initial population of chromosomes and create subsequent generations from the previous generation. The search is performed using three fundamental operators: *reproduction*, *crossover*, and *mutation*. Reproduction randomly copies individual chromosomes into the new generation with a probability increasing with increasing objective function value. After reproduction, crossover randomly combines building blocks of two chromosomes (parents) to produce two new chromosomes (children). One example of crossover is called one-point crossover. One-point crossover selects a single crossover point at random dividing each parent into two parts. The left part of parent1 is combined with the right part of parent2 to create a child. Likewise, the right part of parent1 is combined with the left part of parent2 to create another child. Crossover recombines schema present in the parents and passes them to the children. Mutation randomly changes a chromosome enabling the introduction of schema not present in the population. Although mutation plays a an important role in GAs, it is typically used much less frequently than crossover. High mutation rates are disruptive to the GA. Since crossover and mutation have a random component, they are said to be stochastic operators. GAs are useful since they preserve and recombine building blocks of solutions to make new solutions.

The GA representation of the OLH is an ordering of the objects. The first gene represents the first object in the list, the second gene represents the second object, etc. Each object occurs exactly once in the chromosome. This representation is referred to as a *permutation*. Figure 3 shows an example of a permutation chromosome. The objects are given to First Fit in the order specified by the chromosome. The crossover used in this approach differs from other permutation crossovers. The placement of each object under First Fit depends on the placement of all objects appearing to the left in the chromosome. The crossover preserves schemata of the left side of the parent chromosome and passes them to the children as the left part of the chromosome. The GA uses Equation 7 as the objective function.

**Equation 7**

$$f_{GA} = m + \frac{\alpha}{\sum_{i=1}^{m}\left(\overline{fill} - fill_i\right)^2}$$

$m$: Number of bins

$\alpha$: A positive constant

$fill_i$: The sum of the object sizes of the objects in bin $i$

$$\overline{fill} = \frac{1}{m}\sum_{i=1}^{m} fill_i$$

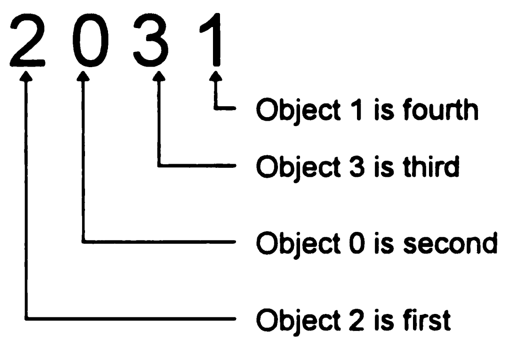


**Figure 3: Permutation Chromosome Representation**

The Open Loop Hybrid Algorithm is composed of the GA and the MAS. The GA is employed for several generations until it produces a solution of desirable quality. The MAS further optimizes the solution to produce the final solution. Figure 4 shows the OLHA.

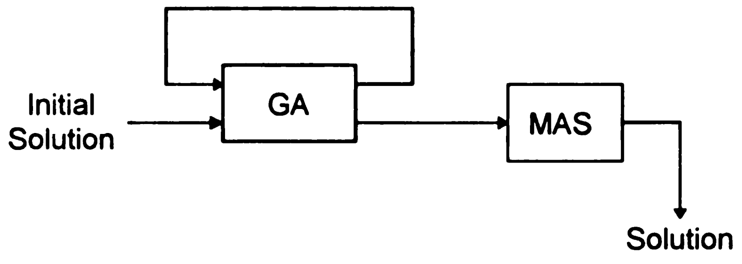


**Figure 4: Open Loop Hybrid Algorithm**

### 4.3.2 The Reduction Method

The Reduction Method is a powerful algorithm that produces near-optimal solutions to the BPP proposed by [Martello and Toth, 90]. In order to describe the algorithm, several definitions are necessary.

**Feasible set:** A *feasible set* of objects is defined as any subset of the objects $W$ whose sizes sum to less than or equal to the bin capacity $C$.

$F$ is a feasible set *iff* $F \subseteq W$ and $\sum_{i \in F} s_i \le C$
$W = \{1,2,\ldots,n\}$ The set of all objects
$s_i$ is the size of object $i$
$C$ is the bin capacity

Consider an instance consisting of a set of objects $W$. If a feasible set $F$ is assigned to a bin, then the instance $W$ is reduced to an instance $W - F$.

**Dominance:** Given two feasible sets $F_1$ and $F_2$, $F_1$ *dominates* $F_2$ *iff* the number of bins for the optimal solution of the reduced instance $W - F_1$ is not greater than the number of bins required for the optimal solution of the reduced instance $W - F_2$.

**Dominance criterion:** Given two distinct feasible sets $F_1$ and $F_2$, if there exists a partition $P = \{P_1,\ldots,P_l\}$ of $F_2$ and a subset $\{i_1,\ldots,i_l\}$ of $F_1$ such that $s_{i_h} \ge \sum_{k \subseteq P_h} s_k$ for $h = 1,\ldots l$, then $F_1$ dominates $F_2$. Figure 5 clarifies this definition.

Checking if one feasible set $F_1$ dominates another feasible set $F_2$ requires either solving the reduced instances $W - F_1$ and $W - F_2$ or using the dominance criterion. Solving the reduced instances requires exponential-time. Using the dominance criterion is computationally less intensive since only the objects in the two feasible sets are examined.

The Reduction Method makes use of the dominance criterion. This procedure reduces the size of an instance of BPP by considering all feasible sets, finding one (call it $F$) dominating all of the others, assigning $F$ to a new bin and removing $F$ from the set of remaining objects $W$. Since that leaves fewer objects to consider $(W - F)$, the problem is *reduced*. The cardinality of the feasible sets considered are limited and checking for dominance is done only through the dominance criterion. If no set dominating all others can be found, the problem is relaxed by removing the smallest

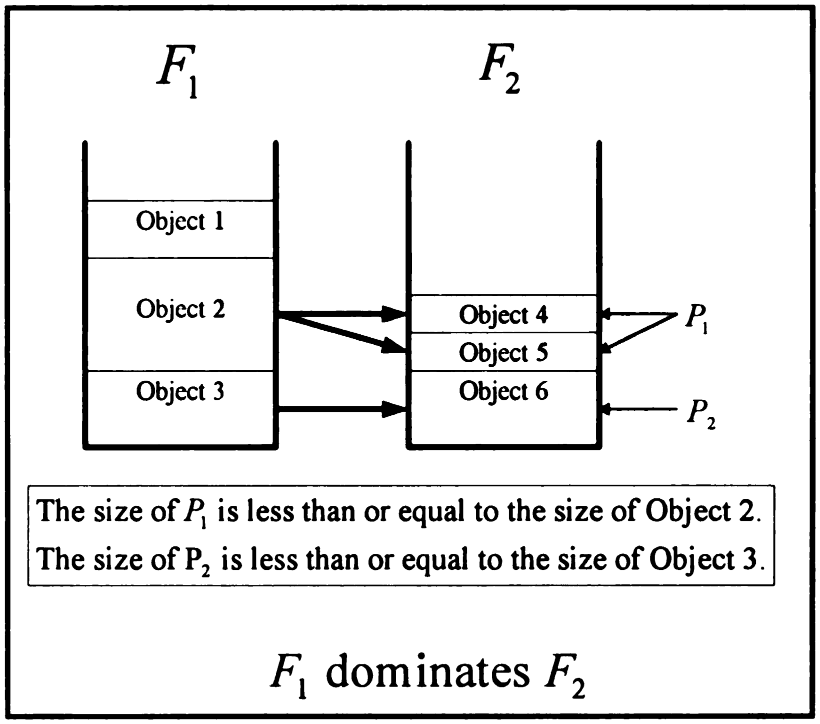

$F_1$
$F_2$
Object 1
Object 2
Object 3
Object 4
Object 5
Object 6
$P_1$
$P_2$
The size of $P_1$ is less than or equal to the size of Object 2.
The size of $P_2$ is less than or equal to the size of Object 3.
$F_1$ dominates $F_2$

**Figure 5: Dominance Criterion**

object among those which are not yet assigned to any bin. These relaxed objects are later inserted with a different method such as FFD.

### 4.3.3 The Hybrid Grouping Genetic Algorithm

Traditional GAs are ill-suited for grouping problems. Consider the representation using one gene per object as shown in Figure 6. Chromosomes using this representation exhibit redundancy. Many chromosomes describe the same solution. For example, the chromosomes AABB and BBAA have a group comprised of objects 0 and 1 and a group comprised of objects 2 and 3. Since the names of the groups are irrelevant, the two chromosomes are equivalent. The redundancy can grow exponentially as the number of objects increases. An exponential factor of redundancy drastically enlarges the search space. Traditional crossover, such as one-point crossover, does not pass meaningful information from parents to children. In the example of Figure 7, none of the groups in the children exist in the parents. Infeasible solutions (overfilled bins) and poor solutions dominate the search-space.

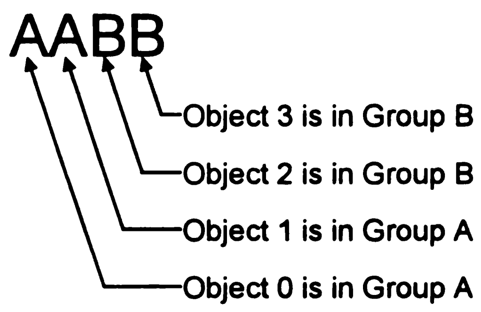


**Figure 6: Traditional Chromosome Representation**

**Parent 1: ABC|ABC**
**Parent 2: AAB|BCC**
**Child 1: ABCBCC**
**Child 2: AABABC**

**Figure 7: Traditional Crossover**

Emanuel Falkenauer proposes a particular adaptation of GAs for grouping problems that is better suited for grouping problems than traditional GAs [Falkenauer, 92],[Falkenauer, 94A],[Falkenauer, 94B],[Falkenauer, 95]. He developed one of the most powerful methods known for solving the BPP, and it provides the basis for our work. Falkenauer identifies the schemata of grouping problems as the groups. He defines a representation, crossover operator, and a mutation operator that work with groups as the building blocks. He calls the modified GA a Grouping Genetic Algorithm (GGA). Crossover and mutation produce children with each object assigned to exactly one bin and no bin is overfilled. The GGA, therefore, is restricted to working only with feasible solutions.

The GGA's chromosomes are composed of two parts: the *object part* and the *group part*. The object part has one locus for each object. The object part identifies which objects form which group. In Figure 8, the first object (Object 0) is in the group specified by the first gene (Group A). The group part lists each group exactly once in any order. Crossover and mutation work with the group part directly and use the object part indirectly. The group part is necessary to allow linkage among the groups under crossover. Groups listed near each other in the group part of the chromosome are more likely to remain together after crossover. In Figure 8, the chromosome represents a configuration with three groups. The first group (Group B) is "closer" to the second group (Group A) than it is to the third group (Group C), in strength of linkage. Groups having strong linkage are more likely to remain together under crossover. Since the number of groups is not constant, the chromosomes can have different lengths.

The goal of the GGA's crossover is to pass possibly linked groups from parents to the children. Crossover is performed as follows [Falkenauer, 94A]:

1) Select at random two crossing sites in the group part, delimiting the *crossing*

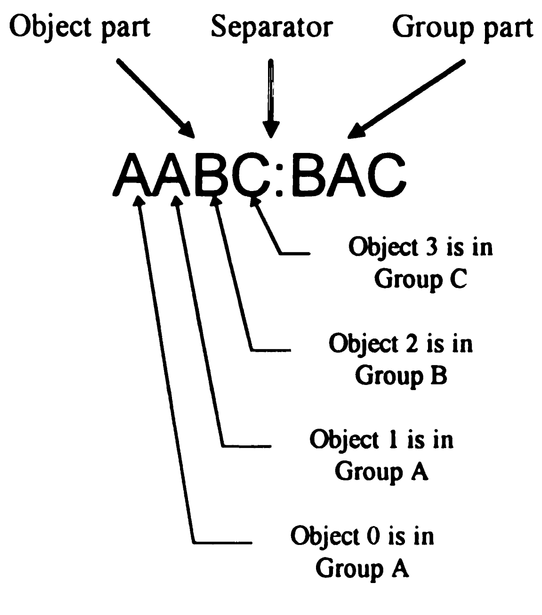


**Figure 8: Chromosome Representation of the Grouping Genetic Algorithm**

*section*, in each of the two parents.

2) *Inject* the contents of the crossing section of Parent 1 at the first crossing site of the Parent 2. This means injecting some of the groups from Parent 1 into Parent 2.
3) Eliminate all items now occurring twice from the groups of which they were members in Parent 2. The 'old' membership of these items gives way to the membership specified by the 'new' injected groups. Some of the 'old' groups coming from the second parent are thus altered: they do not contain all the same items anymore, since some of those items had to be eliminated. Remove these groups from the group part of Parent 2 and place objects belonging to these groups in a queue of objects lacking membership in a group.
4) Assign each object in the queue to a group. The algorithm used is problem-specific. For the BPP, the objects are reinserted using FFD.
5) Perform Steps 2 through 4 on the two parents with their roles reversed in order to generate the second child.

Example: Groups from Parent 1 are injected into Parent 2. Parent 2 is represented by small letters so that it can be differentiated from Parent 1.

**(Parent 1) AABCC:ACB**

**(Parent 2) abcbc:cab**

After step 1:

**(Parent 1) AABCC:|A|CB**

**(Parent 2) abcbc:c|ab|**

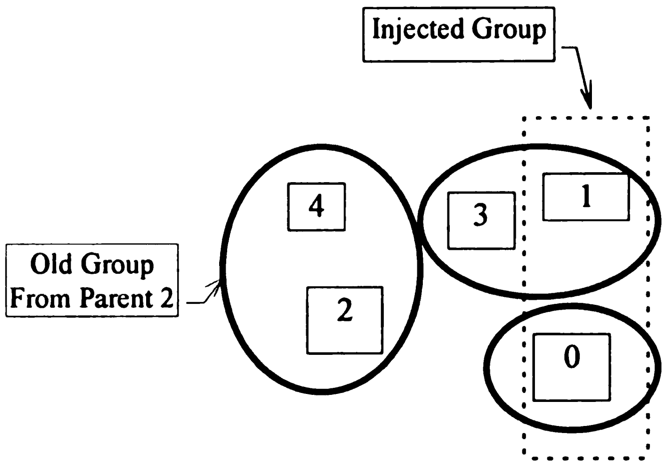


**Figure 9: Crossover Example**

After step 2:

**(Parent 1) AABCC:|A|CB**

**(Parent 2) abcbc:cAab**

After step 3:

**(Parent 1) AABCC:|A|CB**

**(Parent 2) AAc⊗c:cA**

**Queue = {object 2}**

Mutation in the GGA is performed as follows:

1) Select at random several groups to eliminate.
2) Place the objects in these groups into the queue.
3) Reinsert the objects in the queue into the solution. The algorithm used is problem-specific. For the BPP, the objects are reinserted using FFD.

Falkenauer uses Equation 8 as the objective function for the BPP. Since the GGA's crossover and mutation operators create only feasible solutions, the objective

**Equation 8**

$$f_{BPP} = \frac{\sum_{i=1}^{m} fill_i^2}{C^2 m}$$

$m$: the number of bins being used

$fill_i$: the sum of the sizes of the objects in $bin_i$

$C$: the bin capacity

function does not need a penalty factor to discourage constraint violations.

[Falkenauer, 92] and [Falkenauer, 94B] present results on the BPP as a graph. Falkenauer varied the difficulty of the problems by selecting object sizes so that an optimal solution can have some empty space remaining in each bin. The empty space is called *leeway*. Although the GGA outperformed FFD, better methods for the BPP exist such as the Reduction Method. Since the GGA utilizes FFD in crossover and mutation, Falkenauer compares the performance of the GGA to that of FFD to show that the better performance of GGA is the result of processing schemata.

Falkenauer later added a heuristic hill-climber inspired by the Reduction Method to the GGA. He calls the new system the Hybrid Grouping Genetic Algorithm (HGGA).

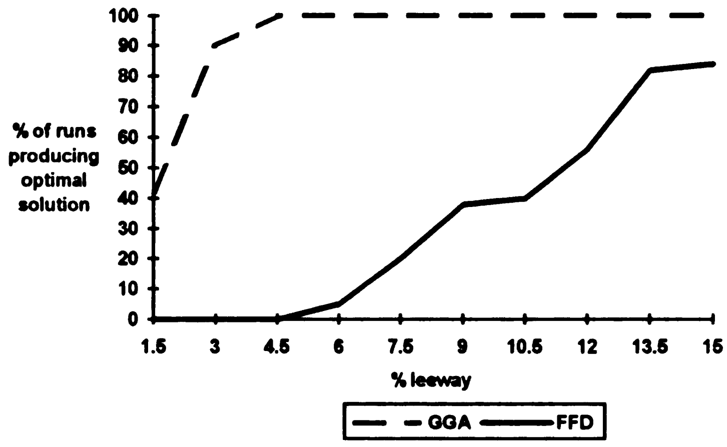
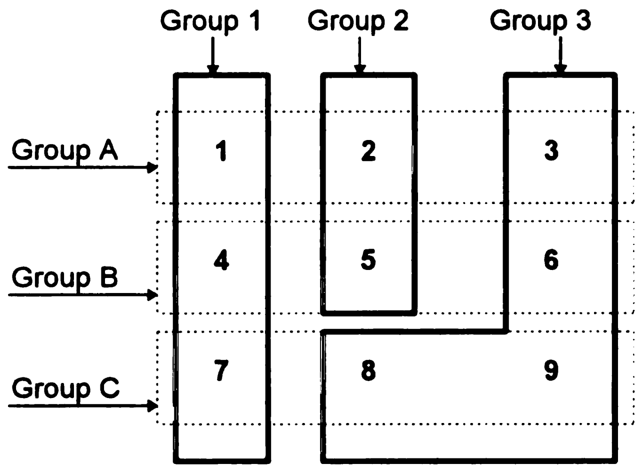


**Figure 10: GGA vs. FFD**

The HGGA produces far more competitive results with the addition of the heuristic hill-climber. The heuristic is valid only for the BPP. The heuristic is added to the crossover and mutation operators. In both operators, the heuristic is inserted before using FFD to reinsert the objects from the queue. The heuristic performs a local optimization as follows: For each bin already in the solution, replacements of up to three objects in the bin by one or two objects in the queue are performed if they make the bin fuller without exceeding the bin capacity.



| Preservation of Group | Inhibits Groups |
|---|---|
| A | 1,2,3 |
| B | 1,2,3 |
| C | 1,3 |
| 1 | A,B,C |
| 2 | A,B |
| 3 | A,B,C |

**Figure 11: Example Demonstrating Mutation-like Behavior of the GGA's Crossover**

The GGA's crossover operator exhibits a mutation-like behavior. Crossover combines schemata from two parents to form children. The GGA's crossover uses groups as the schemata. Unless the two parents are similar, the objects in a particular group in parent1 are spread among several groups in parent2.

Figure 11 illustrates that the preservation of each group in one parent can inhibit the preservation of many groups from the other parent. Only a small portion of the groups present in the children can be obtained from the parents. A heuristic, such as FFD in the case of the BPP, assembles the remainder of the groups. Since a large portion of the schemata in the children is not present in the parents, the crossover operator exhibits a mutation-like property. Figure 11 demonstrates that the preservation of groups in one parent can inhibit many groups in the other parent. Any crossover operator that works with groups as schemata exhibits this mutation-like behavior.

# 5. The Swapping Heuristic Applied to the Bin Packing Problem

## *5.1 Rationale for the Swapping Heuristic*

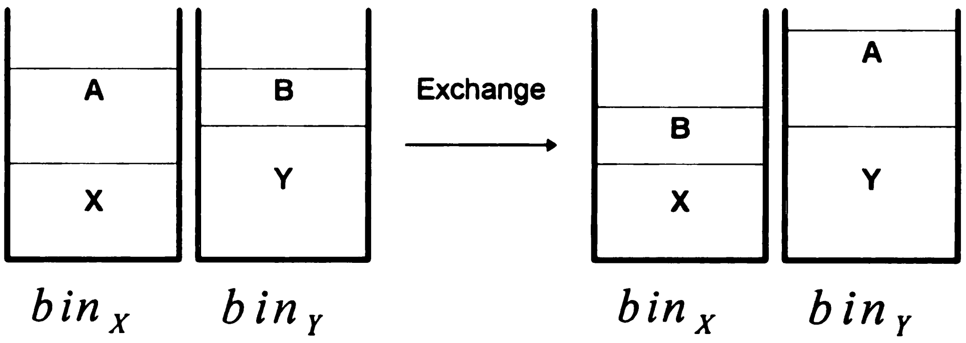
The Swapping Heuristic (SH) is the product of extensive work with Falkenauer's HGGA. The solutions obtained with the HGGA are simply not attainable without the use of the Martello and Toth Heuristic. The heuristic, therefore, must be processing domain specific knowledge to help produce these solutions. Is the Martello and Toth Heuristic the best way to utilize this additional knowledge? This question began the work leading to the SH. The SH was designed as another heuristic for use in the HGGA either in conjunction with the Martello and Toth Heuristic or as a replacement. The SH became powerful enough to produce quality solutions on its own, making the GGA unnecessary.

Falkenauer's objective function (Equation 9) provided the GGA a metric for guiding the search. The same metric can be utilized directly by a local search technique. Exchanges can sometimes be found between two bins such that one bin after the exchange is fuller than either bin was before the exchange. These exchanges always increase the value of Falkenauer's objective function. To prove this, there are two possible cases to consider: the number of bins remains the same (Case 1) and the number of bins decreases by one (Case 2).

**Equation 9**

$$\text{Falkenauer's objective function for BPP: } \frac{\sum_{i=1}^{m} fill_i^2}{C^2 m}$$

### 5.1.1 Case 1: The Number Of Bins Stays The Same



X: the set of objects staying in $bin_x$.
Y: the set of objects staying in $bin_Y$
A: the set of objects moving from $bin_X$ to $bin_Y$
B: the set of objects moving from $bin_Y$ to $bin_X$

**Figure 12: Case 1 Exchanges**

Under what conditions is $f_{BPP}(\text{Before Exchange}) < f_{BPP}(\text{After Exchange})$?

$$\left(fill_X^2 + fill_Y^2 + \sum_{\substack{i=1\\ i\neq X\\ i\neq Y}}^{m} fill_i^2\right)\frac{1}{C^2 m} < \left(fill_X^2 + fill_Y^2 + \sum_{\substack{i=1\\ i\neq X\\ i\neq Y}}^{m} fill_i^2\right)\frac{1}{C^2 m}$$

$$fill_X^2 + fill_Y^2 < fill_X^2 + fill_Y^2$$

Before Exchange: $fill_X = (x+a),\ fill_Y = (y+b)$

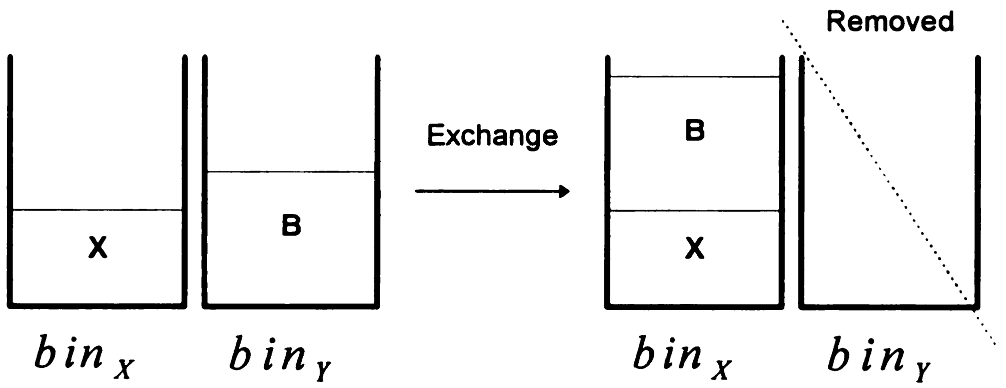
After Exchange: $fill_X = (x+b), fill_Y = (y+a)$

$$(x+a)^2 + (y+b)^2 < (x+b)^2 + (y+a)^2$$

$$\cancel{x^2} + 2ax + \cancel{a^2} + \cancel{y^2} + 2by + \cancel{b^2} < \cancel{x^2} + 2bx + \cancel{b^2} + \cancel{y^2} + 2ay + \cancel{a^2}$$

$ax + by < bx + ay$
$ax - ay < bx - by$
$a(x - y) < b(x - y)$
Dividing both sides by $(x - y)$
If $(x - y) > 0$ then the value of the objective function increases for $a < b$
If $(x - y) < 0$ then the value of the objective function increases for $a > b$
If a resulting bin is filled more than both starting bins, the objective function always increases.

### 5.1.2 Case 2: The Number Of Bins Decreases



X: the set of objects staying in $bin_x$.
Y: the set of objects staying in $bin_Y$
A: the set of objects moving from $bin_X$ to $bin_Y$
B: the set of objects moving from $bin_Y$ to $bin_X$

**Figure 13: Case 2 Exchanges**

Under what conditions is $f_{BPP}(\text{Before Exchange}) < f_{BPP}(\text{After Exchange})$?

$$\left( fill_X^2 + fill_Y^2 + \sum_{\substack{i=1 \\ i \neq X \\ i \neq Y}}^{m} fill_i^2 \right) \frac{1}{C^2 m} < \left( fill_X^2 + \sum_{\substack{i=1 \\ i \neq X}}^{m} fill_i^2 \right) \frac{1}{C^2 (m-1)}$$

$$\left( fill_X^2 + fill_Y^2 \right) \frac{1}{m} < \left( fill_X^2 \right) \frac{1}{m-1}$$

Before Exchange: $fill_X = x,\ fill_Y = b$
After Exchange: $fill_X = (x + b),\ bin_Y$ is removed.

$$\frac{(x^2+b^2)}{m} < \frac{(x+b)^2}{m-1}$$
$$\frac{x^2+b^2}{m} < \frac{x^2+2xb+b^2}{m-1}$$

Value of Falkenauer's objective function increases for all $x$ and $b$.
($x$ and $b$ are positive values)

Falkenauer utilizes a genetic algorithm to optimize bin packing solutions relative to the objective function. It is shown that exchanges of objects between groups can be identified that increase the value of the objective function. Pairs of bins can be examined until no more of these exchanges can be found. This serves as the basis for the SH.

## *5.2 Description of the Swapping Heuristic Applied to the Bin Packing Problem*

Section 5.1 shows that gains which result from exchanges of objects between two bins can be used to optimize a solution to the BPP. The objective function value increases if, after an exchange, a resulting bin is fuller than each of the starting bins. Any solution to the Bin Packing Problem can possibly be made better by performing a local optimization. The local search is done by performing exchanges between any two bins such that after the exchange, one bin is fuller (the other is less full) than either bin was before the exchange. The less full bin often benefits the overall solution since it is easier to eliminate by moving the objects into other bins and has added room to accommodate more objects making the elimination of another bin easier. The SH can start with any solution and improve it. Thus, it can be used to enhance the performance of any other bin packing algorithm.

An iteration is a pass through every pair of bins. The stopping criterion is a limit to the number of iterations performed in which no beneficial exchange has been

```
start with any initial solution

counter=0;
while (counter<number_of_tries)
  {
    counter++;
    for ($i = 1$; $i < m - 1$; $i++$)
      for ($j = i + 1$; $j < m - 1$; $j++$)
        {
          exchange($bin_i$,$bin_j$)
          if progress has been made then
             counter=0; /* Reset Counter */
        }
  }

void exchange($bin_i$,$bin_j$)
1. Let O be the union of the set of objects from $bin_i$ and the set
   of objects from $bin_j$.
2. Compute the sizes of the elements of all binary partitions of
   $O$ which do not exceed $C$.
3. Select a partition: If there is exactly one partition from
   step 2, select it.  If there are more than one partition from
   step 2, select one at random.
4. Using the selected partition from 4, place the objects in the
   larger element in $bin_i$, and the objects in the smaller element
   in $bin_j$.
```

**Figure 14: Basic Description of the Swapping Heuristic**

performed. This allows the heuristic to continue working as long as it continues making progress. In our experiments, we used twenty as the limit. The basic algorithm for the SH is given in Figure 14.

The function *exchange* takes two bins and determines the combination of objects from the two bins with the maximum sum that is less than or equal to the bin size. This is accomplished by computing all of the possible sums the objects can produce. This requires $2^z$ sums where $z$ is the number of objects in the two bins. This requires exponential time with respect to $z$. This is practical only for instances with small $z$. In order to produce a heuristic of reasonable complexity when $z$ is large, several objects are treated as one, putting an upper limit on the number of sums computed. Figure 14 describes the exchange function.

The SH can get "stuck" on a non-optimal solution. This is commonly caused by many small objects forming a well filled bin too early. The objects in a well-filled bin are difficult to break apart. Small objects are often the easiest to pack since they fit in the spaces left after packing the large objects. The SH can prevent many small objects from forming a well-packed bin too early by first limiting the number of objects allowed in a bin and then relaxing that restriction in a stepwise fashion. Figure 15 incorporates the object limit into the SH.

```
Calculate or read final_object_limit, number_of_tries
Place each object in its own bin

for(object_limit=2; object_limit<=final_object_limit; object_limit++)
  counter=0;
  while(counter<number_of_tries)
    {
      counter++;
      for(i=1; i<=number_of_bins-1; i++)
        for(j=i+1; j<=number_of_bins; j++)
          {
             large=$\max\{bin_i, bin_j\}$;
             exchange($bin_i, bin_j$);
             if($\max\{bin_i, bin_j\}$>large)
                    counter=0;
          }
    }

void exchange($bin_i, bin_j$)
1. Let O be the union of the set of objects from $bin_i$ and the set
   of objects from $bin_j$.
2. Compute the sizes of the elements of all binary partitions of
   $O$ which do not exceed $C$ and the number of objects in the
   element satisfies the following condition:
   (number of objects in $O$ − object limit) ≤ (number of objects) ≤ (object limit)
1. Select a partition: If there is exactly one largest partition
   from step 2, select it.  If there is more than one largest
   partition from step 2, select one at random.
2. Using the selected partition from step 3, place the objects
   in the larger element in $bin_i$ and the objects in the smaller
   element in $bin_j$.
```

**Figure 15: Swapping Heuristic with Object Limit**

## *5.3 Worst-Case Performance*

Research involving algorithms focuses on determining and proving bounds on worst-case performance. This includes the worst-case quality of solution and the wost-case execution time. The worst-case solution is expressed in terms of the optimal solution. Figure 16 shows the instance giving rise to the currently known worst-case solution for the SH. The objects in the optimal arrangement require three bins. The objects in the arrangement produced by the SH require four bins. If this instance is proven to be the worst-case example for the SH, the performance of the SH would obey Equation 10. The worst-case performance, if proved, would be better than that of FFD and other simple methods but not the approximation schemes. Since the SH searches using a single point, the SH working alone can be deceived by such an instance. The SH working in conjunction with a GA can search more thoroughly and would less likely be deceived by such an instance.

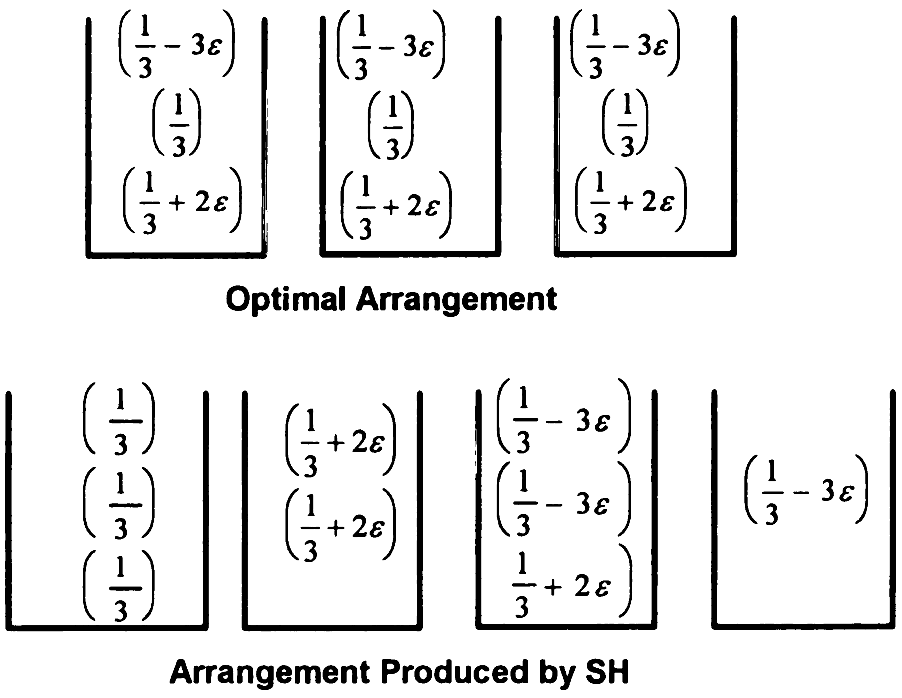


Figure 16: Worst Known Case for the SH

**Equation 10**

$$SH(I) \le \frac{13}{12} Opt(I) + 1$$

$SH(I)$: Number of bins required by the SH for instance $I$

$Opt(I)$: Minimum number of bins required for instance $I$

The worst-case execution time is expressed in terms of the size of the instance. The size of an instance of BPP is $n$, the number of objects. The SH's counter makes determining the worst-case execution time difficult. Several runs have been performed to provide some indication of how the execution time increases as the number of objects increases. The runs use objects selected in the range [1,10000] and have bin capacity 10000. The execution times are the product of two parts: the number of iterations and the average time required for each iteration. Figure 19 shows the execution time per iteration of the runs to increase as a function of $n^2$. The number of bins grows proportionally to the number of objects for the same bin capacity. An iteration involves working with every pair of bins. For $m$ bins, $\frac{(m)(m-1)}{2}$ pairs of bins are analyzed in each iteration. Therefore, the time for each iteration is expected to increase as a function of $n^2$. The number of iterations required are experimentally shown to increase approximately linearly as the number of objects increases. The third column in Table 1 shows that the number of iterations divided by the number of objects generally decreases and the number of objects increases.

Table 1: Execution Times for Increasing Number of Objects

| Number of Objects | Number of Iterations | $\frac{\text{Num of Iterations}}{n} \times 1000$ | Time Sparc 10 CPU seconds | $\frac{\sqrt[3]{\text{Time}}}{n} \times 1000$ |
|---|---|---|---|---|
| 1000 | 119 | 119 | 267 | 6.44 |
| 2000 | 135 | 67.5 | 1223 | 5.35 |
| 3000 | 167 | 55.7 | 3461 | 5.04 |
| 4000 | 217 | 54.3 | 8077 | 5.02 |
| 5000 | 347 | 69.4 | 20259 | 5.45 |
| 6000 | 282 | 47.0 | 24116 | 4.82 |
| 7000 | 336 | 48.0 | 39340 | 4.86 |
| 8000 | 311 | 38.9 | 47922 | 4.54 |

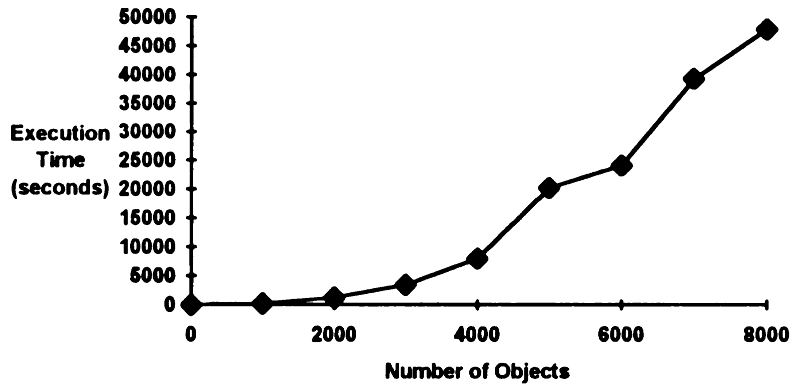


Figure 17: Execution Time vs. Number of Objects

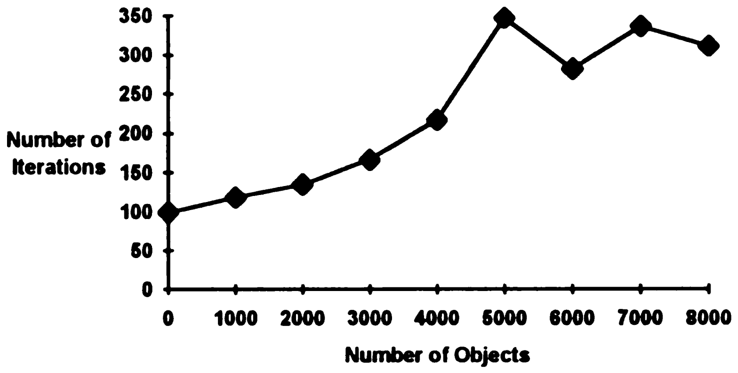


Figure 18: Number of Iterations vs. Number of Objects

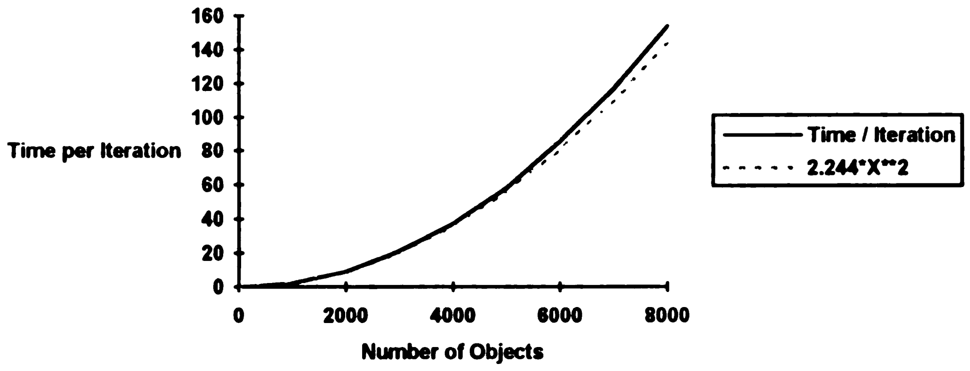


Figure 19: Time per Iteration vs. Number of Objects

## *5.4 Results: Bin Packing Problem*

### 5.4.1 Comparison to Standard Methods

Table 2 compares the performance of the SH to that of FF, FFD, BF, and BFD. The runs use 1000 objects with integer object sizes selected uniformly random in the given range. Equation 11 provides a lower bound for runs 1 through 9. The performance bound of FFD (Equation 2) provides a tighter lower bound for run 10. These lower bounds may not be achievable.

**Table 2: Results, Bin Packing Problem, Comparison to Standard Methods**

| | | | Number of Bins | | | | | |
|---|---|---|---|---|---|---|---|---|
| Run | Range of Object Sizes | Bin Capacity | First Fit | First Fit Des | Best Fit | Best Fit Des | SH | Lower Bound |
| 1 | [1,1000] | 2000 | 251 | 247 | 251 | 247 | 247 | 247 |
| 2 | [1,1000] | 2500 | 200 | 198 | 200 | 198 | 198 | 198 |
| 3 | [1,1000] | 3000 | 166 | 165 | 166 | 165 | 165 | 165 |
| 4 | [200,1000] | 2000 | 309 | 302 | 308 | 302 | 297 | 297 |
| 5 | [200,1000] | 2500 | 245 | 240 | 245 | 240 | 238 | 238 |
| 6 | [200,1000] | 3000 | 203 | 201 | 203 | 201 | 198 | 198 |
| 7 | [500,1000] | 2000 | 422 | 417 | 422 | 417 | 400 | 375 |
| 8 | [500,1000] | 2500 | 323 | 313 | 323 | 313 | 300 | 300 |
| 9 | [500,1000] | 3000 | 268 | 267 | 268 | 267 | 250 | 250 |
| 10 | [800,1000] | 2500 | 474 | 457 | 474 | 457 | 446 | 371 |

**Equation 11**

$$\text{Lower Bound} = \left\lceil \frac{\sum_{i=1}^{n} s_i}{C} \right\rceil$$

## 5.4.2 Comparison to Reduction Method and Hybrid Grouping Genetic Algorithm

Martello and Toth tested the Reduction Method using various ranges and bin capacities and found instances with bin capacity $C = 150$ and integer object sizes selected uniformly random in the range [20,100] to be the most difficult. Falkenauer, however, wanted to know the optimal number of bins for each instance. He accomplished this by using a procedure that selects object sizes so that they exactly fill the bins. Most object sizes are selected uniformly random in the range [20,100]. Some object sizes are selected in the range [20,100] so that they fill the bins completely leaving no space. All bins are full except possibly the last. Figure 20 describes the procedure in detail. Falkenauer generated 20 instances of the 'Uniforms' with 1000 objects.

For each instance, Falkenauer ran the HGGA until it obtains the optimal number of bins and recorded the time. The SH , similarly, was run until it reaches the optimal number of bins. The HGGA, like the SH, is not guaranteed to determine the optimal number of bins in a finite amount of time.

Table 3 and Table 4: use the following:

- Theo: Theoretically minimum number of bins required for the instance
- Loss: Number of bins in solution minus theoretically minimum number of bins
- Eval: Number of objective function evaluations performed by the HGGA.
- Time: Execution time in seconds on the machine indicated by footnotes.
- Backs: Number of iterations performed during the reduction method.
- Passes: Number of passes through the bins.
- Speed-Up: HGGA execution time divided by the SH execution time

```
$C = 150$;   /* C is the Bin Capacity */
$S = C$;   /* Initialize S, the available to the bin capacity */
for($i = 1$, $i <= n$, $i++$)
  {
    repeat
      $R$= random number generated in the range [20,100];
    until ($R \geq S$) or ($S - R \geq 20$)
    if ($R \geq S$)
      {
        $s_i = S$;
        $S = C$;
      }
    if ($S - R \leq 20$)
      {
        $s_i = R$;
        $S = S - R$;
      }
  }
```

**Figure 20: Procedure for Creating Instances of the 'Uniforms'**

Table 3 compares the executions times of the SH and the HGGA on the 'Uniforms' with 1000 objects. The SH is executed with the following parameter settings on the 'Uniforms': Starting object limit = 6, Final object limit=6, number of tries = 20. The execution times listed for the Reduction method and the HGGA in Table 3 are obtained from [Falkenauer, 94A]. The instances are those created by Falkenauer.

Falkenauer identifies the 'Triplets' as another difficult set of bin packing instances. The 'Triplets' object sizes are selected so that three objects exactly fill a bin. The first object size is drawn uniformly from the range [380,490]. That leaves a space $S$ remaining in the bin. The second object is drawn uniformly from [250,$S$/2). The third object is chosen so that it completely fills the remaining space in the bin, leaving no leeway. These data sets are designed to be extremely difficult bin packing problems. Falkenauer generated 20 instances of 'Triplets' of 501 objects. Table 4 compares the execution times of the SH and the HGGA on the 'Triplets' instances with 501 objects. The 'Triplets' are run with the following parameters: starting object limit = 2, final object limit = 6, number of tries = 20.

The instances used in Table 3 and Table 4 are part of a set of Operations Research benchmarks maintained at the Imperial College of Management (http://mscmga.ms.ic.ac.uk/info.html). These benchmark instances are available to all researchers to provide a way to compare methods. To date, the SH is the only known method capable of solving these instances in less time than the HGGA, and the time difference is very large.

**Table 3: Results, Bin Packing Problem, 'Uniforms'**

| Runs | | HGGA | | | Reduction Method | | | SH | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Theo | Loss | Evals | Time R4000 CPU seconds | Loss | Backs | Time R4000 CPU seconds | Loss | Passes | Time R4000 CPU seconds | Speed-up |
| 1 | 399 | 0 | 2211 | 2924.7 | 4 | 3.5M | 3279.0 | 0 | 2 | 16.78 | 174.3 |
| 2 | 406 | 0 | 2948 | 4040.2 | 4 | 5M | 4886.6 | 0 | 2 | 16.73 | 241.5 |
| 3 | 411 | 0 | 4958 | 6262.1 | 5 | 8.5M | 6606.1 | 0 | 2 | 13.81 | 359.0 |
| 4 | 411 | 0 | 35376 | 32714.3 | 5 | 50M | 40285.6 | 0 | 14 | 52.90 | 618.4 |
| 5 | 397 | 0 | 8844 | 11862.0 | 4 | 20M | 20689.8 | 0 | 2 | 18.25 | 650.0 |
| 6 | 399 | 0 | 2948 | 3774.3 | 3 | 5M | 4216.3 | 0 | 2 | 16.80 | 224.7 |
| 7 | 395 | 0 | 2010 | 3033.2 | 3 | 3M | 3449.7 | 0 | 2 | 20.58 | 147.4 |
| 8 | 404 | 0 | 7303 | 9878.8 | 2 | 12.5M | 12674.4 | 0 | 3 | 19.11 | 516.9 |
| 9 | 399 | 0 | 4355 | 5585.2 | 3 | 4.5M | 6874.0 | 0 | 2 | 19.81 | 281.9 |
| 10 | 397 | 0 | 6968 | 8126.2 | 5 | 12.2M | 9568.2 | 0 | 2 | 18.12 | 448.5 |
| 11 | 400 | 0 | 2278 | 3359.1 | 4 | 4M | 3542.8 | 0 | 2 | 20.29 | 165.6 |
| 12 | 401 | 0 | 6700 | 6782.3 | 3 | 8.1M | 7422.4 | 0 | 2 | 15.26 | 444.4 |
| 13 | 393 | 0 | 1943 | 2537.4 | 3 | 3.2M | 2714.0 | 0 | 2 | 20.49 | 123.8 |
| 14 | 396 | 0 | 14137 | 11828.8 | 5 | 20M | 23319.4 | 0 | 2 | 17.41 | 679.4 |
| 15 | 394 | 0 | 5762 | 5838.1 | 5 | 5M | 6770.9 | 0 | 4 | 25.00 | 233.5 |
| 16 | 402 | 0 | 13802 | 12610.8 | 5 | 20M | 20458.4 | 0 | 5 | 24.12 | 522.8 |
| 17 | 404 | 0 | 2278 | 2740.8 | 3 | 3M | 3139.6 | 0 | 2 | 14.83 | 184.8 |
| 18 | 404 | 0 | 2077 | 2379.4 | 3 | 3M | 2506.4 | 0 | 2 | 18.72 | 127.1 |
| 19 | 399 | 0 | 1005 | 1329.7 | 4 | 1.5M | 1353.2 | 0 | 2 | 14.70 | 90.5 |
| 20 | 400 | 0 | 2680 | 3564.2 | 5 | 3M | 4109.6 | 0 | 2 | 14.82 | 240.5 |
| Ave | | | | 7058.6 | | | | | | 19.9 | |

**Table 4: Results, Bin Packing Problem, Triplets Distribution**

| Runs | | HGGA | | | Reduction Method | | | SH | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Theo | Loss | Evals | Time R4000 CPU seconds | Loss | Backs | Time R4000 CPU seconds | Loss | Passes | Time Sparc 10 CPU seconds | Speed -up |
| 1 | 167 | 0 | 3752 | 1806.7 | 17 | 1.5M | 5828.9 | 0 | 56 | 40.62 | 44.5 |
| 2 | 167 | 0 | 3551 | 1582.2 | 14 | 1.5M | 3437.4 | 0 | 87 | 67.18 | 23.6 |
| 3 | 167 | 0 | 1809 | 1234.5 | 10 | 1.5M | 2358.7 | 0 | 44 | 31.87 | 38.7 |
| 4 | 167 | 0 | 3082 | 1821.9 | 13 | 1.5M | 3398.0 | 0 | 94 | 73.49 | 24.8 |
| 5 | 167 | 0 | 5360 | 2355.2 | 14 | 1.5M | 3709.8 | 0 | 81 | 62.76 | 37.5 |
| 6 | 167 | 0 | 2881 | 1424.0 | 16 | 1.5M | 10624.4 | 0 | 89 | 67.70 | 21.0 |
| 7 | 167 | 0 | 1809 | 1161.4 | 16 | 1.5M | 5788.5 | 0 | 119 | 91.80 | 12.7 |
| 8 | 167 | 0 | 2613 | 1503.7 | 16 | 1.5M | 5798.9 | 0 | 164 | 128.68 | 11.7 |
| 9 | 167 | 0 | 3685 | 2138.4 | 10 | 1.5M | 2991.3 | 0 | 52 | 36.78 | 58.1 |
| 10 | 167 | 0 | 3082 | 1550.1 | 18 | 1.5M | 5626.3 | 0 | 51 | 35.98 | 43.1 |
| 11 | 167 | 0 | 2010 | 1052.9 | 12 | 1.5M | 3771.4 | 0 | 68 | 50.12 | 21.0 |
| 12 | 167 | 0 | 2814 | 1334.9 | 11 | 1.5M | 3063.7 | 0 | 128 | 99.41 | 13.4 |
| 13 | 167 | 0 | 3216 | 1502.2 | 20 | 1.5M | 5787.1 | 0 | 33 | 21.17 | 71.0 |
| 14 | 167 | 0 | 5293 | 1951.0 | 14 | 1.5M | 4494.9 | 0 | 148<br>175 | 108.40<br>142.33 | 7.8 |
| 15 | 167 | 0 | 3216 | 1473.9 | 16 | 1.5M | 5929.5 | 0 | 151 | 120.11 | 12.3 |
| 16 | 167 | 0 | 4623 | 2350.6 | 14 | 1.5M | 5306.9 | 0 | 57 | 41.53 | 56.6 |
| 17 | 167 | 0 | 2613 | 1178.8 | 16 | 1.5M | 5522.0 | 0 | 129<br>136 | 110.28<br>115.63 | 5.2 |
| 18 | 167 | 0 | 3551 | 1754.2 | 16 | 1.5M | 6277.2 | 0 | 203 | 160.38 | 10.9 |
| 19 | 167 | 0 | 3484 | 1775.5 | 13 | 1.5M | 4164.2 | 0 | 117 | 93.33 | 19.0 |
| 20 | 167 | 0 | 4288 | 2307.2 | 21 | 1.5M | 6519.4 | 0 | 122<br>52 | 98.92<br>37.23 | 16.9 |
| Ave | | | | 1663.0 | | | | | | 91.8 | |

### 5.4.3 Comparison to the Open Loop Hybrid Algorithm

Bilchev presents results using instances that demonstrate the worst-case performance of First Fit Descending. Five instances are created using $m = 1,2\ldots,5$

$$objectsize(i) = \frac{1}{2} + \varepsilon \quad \text{for } 1 \le i \le 6m$$
$$objectsize(i) = \frac{1}{4} + 2\varepsilon \quad \text{for } 6m < i \le 12m$$
$$objectsize(i) = \frac{1}{4} + \varepsilon \quad \text{for } 12m < i \le 18m$$
$$objectsize(i) = \frac{1}{4} + \varepsilon \quad \text{for } 18m < i \le 30m$$
$$\varepsilon = 0.01$$

**Table 5: Results, Bin Packing Problem, Bilchev Instances 1**

| Number of Objects | First Fit Descending Number of Bins | Multi-agent System Number of Bins | Genetic Algorithm Number of Bins | Swapping Heuristic Number of Bins | Swapping Heuristic Execution Time (Sparc 10 CPU seconds) | Optimal Solution Number of Bins |
|---|---|---|---|---|---|---|
| 30 | 11 | 9.1 | 9 | 9 | 0.25 | 9 |
| 60 | 22 | 18.4 | 18 | 18 | 0.76 | 18 |
| 90 | 33 | 27.6 | 27 | 27 | 1.91 | 27 |
| 120 | 44 | 36.3 | 36 | 36 | 3.06 | 36 |
| 150 | 55 | 45.3 | 45 | 45 | 5.20 | 45 |

The second set of instances Bilchev used to report results have 50 objects with object sizes selected in the range [0.05,0.65], with a resolution of 300. Bilchev gave no reasoning for choosing this configuration. The lower bounds of these instances are determined by Equation 12.

**Equation 12**

$$\text{Lower Bound} = \left\lceil \frac{\sum_{i=1}^{n} s_i}{C} \right\rceil$$

**Table 6: Results, Bin Packing Problem, Bilchev Instances 2**

| Run | Swapping Heuristic Number of Bins Required | Swapping Heuristic Execution Time (Sparc 10 CPU seconds) | Lower Bound Number of Bins Required |
|---|---|---|---|
| 1 | 19 | 0.45 | 19 |
| 2 | 20 | 0.46 | 20 |
| 3 | 20 | 0.47 | 20 |

The SH obtained the optimal solution on these three instances with an execution time under a second. Bilchev achieved the optimal answer on the majority of the runs. Bilchev does not comment on execution times.

# 6. Known Methods for Minimum Makespan Problem

## *6.1 Theoretical Results*

### 6.1.1 Longest Job First

Longest Job First (LJF) is a simple approach to the MMP that produces approximate solutions.

**Longest Job First (LJF):** Sort the jobs in decreasing order. For each job, place the job on the machine with the shortest schedule.

[Graham, 69] proved the LJF always produces a schedule obeying Equation 13.

**Equation 13**

$$LJF(I) \le \left(\frac{4}{3} - \frac{1}{3m}\right) Opt(I)$$

$LJF(I)$: Makespan of scheduled produce by LJF on instance $I$.
$Opt(I)$: Theoretically minimum makespan possible for instance $I$.

### 6.1.2 Multifit Algorithm

The Multifit Algorithm is an iterative algorithm that performs a binary search guided by the well-known First Fit Descending Algorithm for the Bin Packing Problem [Coffman, Garey and Johnson 78]. The Multifit Algorithm computes an upper bound $C_U$ and a lower bound $C_L$ on the makespan. It calculates the midpoint C of the two bounds and determines if the First Fit Descending Algorithm is capable of packing all of the objects into M or fewer bins of capacity C. If FFD(C) succeeds, C is used as the new upper limit. If FFD(C) fails, C is used as the new lower limit. The binary search is continued for a specified number of iterations. Figure 21 defines the Multifit Algorithm. The worst case performance of the Multifit Algorithm, given by Equation 14, is better than that of LJF.

**Equation 14**

$$MA(I) \le \left(\frac{72}{61}\right) Opt(I)$$

$MA(I)$: Makespan of the schedule produced by the Multifit Algorithm on instance $I$
$Opt(I)$: Minimum makespan of instance $I$

```
Calculate the starting lower bound: $C_L = \max\left\{\frac{\sum_{i=1}^{n} p_i}{m},\ \max_{1 \le i \le n}\{p_i\}\right\}$

Calculate the starting upper bound: $C_U = \max\left\{\frac{2\sum_{i=1}^{n} p_i}{m},\ \max_{1 \le i \le n}\{p_i\}\right\}$

$\sum_{i=1}^{n} p_i$ is the sum of all the job lengths.
$\max_{1 \le i \le n}\{p_i\}$ is the maximum job length of the instance.
Perform a binary search with $k$ iterations.
for ($I = 1;\ I \le k;\ I = I + 1$)
  {
    Calculate the midpoint: $C = \frac{C_L + C_U}{2}$
    if ($FFD(C) \le m$)
        $C_U = C;$
    else
        $C_L = C;$
  }
```

**Figure 21: The Multifit Algorithm**

## *6.2 Dual Approximation Schemes*

The Dual Approximation Schemes [Hochbaum and Shmoys, 87] provide a polynomial time algorithm with performance given by Equation 15 for any bounded relative error $\varepsilon$.

**Equation 15**

$DAA(I) \le (1+\varepsilon)Opt(I)$

$DAA(I)$: Makespan of the schedule produced by the Dual Approximation Algorithm

$\varepsilon$: Relative error

$Opt(I)$: Minimum makespan of instance $I$.

Although these algorithms do run in polynomial time, they have computation complexity of $O\left(\left(\frac{n}{\varepsilon}\right)^{\frac{1}{\varepsilon^2}}\right)$ for relative error at most $\varepsilon$. For relative error of at most 10% $\varepsilon = 0.1$, their algorithm would run in $O(n^{100})$. For relative error of at most 5%, their algorithm would run in $O(n^{400})$. Although these algorithms have polynomial execution times for any given bounded error, these algorithms are not practical for large $n$. Their paper outlines the implementation of two methods, $\varepsilon = \frac{1}{5}$ and $\varepsilon = \frac{1}{6}$. The latter algorithm is non-trivial to implement and its worst case bound is only slightly better than the Multifit Algorithm's. The implementation of the algorithms with smaller $\varepsilon$ are even more difficult.

## *6.3 Practical Methods*

Simulated annealing is a common practical search tool. [Rao and Iyengar, 94] published results on a variation of the MMP using simulated annealing. Rather than trying to minimize the makespan, they minimize the cost function given by Equation 16.

Simulated Annealing initializes the temperature parameter $T$ to a high value $T_\infty$. The temperature parameter is gradually decreased until a small enough value for the temperature is reached. At each temperature, the system is perturbed several times. At the end of each set of perturbations, the new configuration is accepted or rejected. The new configuration is always accepted if it has a lower cost function. If

**Equation 16**

$$f_{sa} = \sum_{j=1}^{m} (t_j - \bar{t})^2$$

*m*: The number of machines

$t_j$: The sum of the processing times of jobs scheduled to machine *j*

$$\bar{t} = \frac{1}{m} \sum_{i=1}^{m} p_i$$

$p_i$: Processing time of job *i*

**Equation 17**

$$P_{accept} = e^{-\Delta C / T}$$
$$\Delta C = C_{final} - C_{initial}$$

the configuration has a higher cost function, the configuration is accepted with probability that decreases by the amount the cost function increases. The probability is given by Equation 17.

A new configuration is generated by pertubating the current configuration. Pertubations for this problem are done using one of these two methods:

1. Randomly select a job. Relocate the job from the machine in which it is currently assigned to a randomly selected machine. These exchanges coursely optimize the solution.
2. Randomly select two jobs currently assigned to different machines. Exchange the assignments of the two jobs. These exchanges make fine adustments to the solution.

```
Generate a Random Configuration C
$T = T_\infty$
While $T > T_0$ do
  {
    repeat
      {
        generate new configuration $C'$;
        $\Delta C = f(C') - f(C)$;
        $\eta$ = uniformly random number in the range [0,1)
        if $(\Delta C < 0 \; or \; \eta < e^{-\Delta C/T})$
          $C = C'$;
      }
    until thermal equilibrium is reached;
    $T = F(T)$;
  }
output $C$;
```

**Figure 22: Description of the Simulated Annealing Algorithm**

# 7. The Swapping Heuristic applied to the Minimum Makespan Problem

## *7.1 Rationale for the Swapping Heuristic*

Section 5.1 provided the basis for applying the SH to the BPP. An approach similar to the one used for the BPP exists for the MMP. In order to reduce the makespan of a schedule, the longest individual machine schedule must be made shorter by rescheduling jobs. Thus, a reasonable strategy is to identify the machines with the longest schedules and attempt to shorten their schedule by exchanging jobs with other machines. This strategy provides the basis for the application of the SH to the MMP.

## *7.2 Description of the Swapping Heuristic Applied to the Minimum Makespan Problem*

The SH for the MMP is similar to the SH for the BPP. The SH uses a counter as the stopping criterion. If a successful exchange is performed, the counter is reset. A successful exchange is an exchange between two machines such that at least one machine has a schedule of length $t_i = Makespan$ before the exchange and both machines have a schedule of length $t_i < Makespan$ after the exchange. Resetting the counter allows the heuristic to continue working as long as it makes progress. The description of the SH applied to the MMP is provided in Figure 23.

```
Use Longest Job First Heuristic or any other method to produce starting
solution.

counter=0;
while (counter<=number_of_tries)
  {
    counter++;
    makespan=determine_makespan();
    for (i=1; i<=number_of_bins-1; i++)
      for (j=i+1; j<=number_of_bins; i++)
        {
          if( $\max\{machine_i, machine_j\}$ ==makespan)
            {
              exchange($machine_i, machine_j$);
              if( $\max\{machine_i, machine_j\}$ <makespan)
                   counter=0;
            }
        }
  }

void exchange($machine_i, machine_j$)
1. Let O be the union of the set of jobs from $machine_i$ and the set of
   jobs from $machine_j$.
2. Find the binary partition(s) of $O$ such that the size of both
   elements is less than the current makespan.  If no such partitions
   exist, find the partition(s) such that the size of both elements is
   less than or equal to the current makespan.  There exists at least
   one partition.
3. Select a partition: If there is exactly one partition from step 2,
   select it.  If there is more than one partition from step 2, select
   one at random.
4. Using the partition from step 3, place the jobs in the larger element
   on $machine_i$ and the jobs from the smaller element on $machine_j$.
```

**Figure 23: The Swapping Heuristic for the Minimum Makespan Problem**

## *7.3 Results: Minimum Makespan Problem*

### 7.3.1 Comparison to Longest Job First and the Multifit Algorithm

The SH currently has no theoretical worst-case performance results. Table 7 compares the performance of the SH to LJF and the Multifit Algorithm. The runs use instances with 1000 jobs with processing times selected uniformly from the job ranges specified in the table. Equation 18 provides a lower bound for all the runs. An analysis of the instance bounds run 22 tighter. These lower bounds might not be reachable. These runs showed the SH to be dramatically better than LJF and MA.

**Equation 18**

$$\text{Lower Bound} = \left\lceil \frac{\sum_{i=1}^{n} p_i}{m} \right\rceil$$

**Table 7: Results, Minimum Makespan Problem, Comparison to Standard Methods**

| | | | Makespan | | | |
|---|---|---|---|---|---|---|
| Run | Job Processing time Range | Number of Machines | Longest Job First | Multifit Algorithm | Swapping Heuristic | Lower Bound |
| 1 | [1,1000] | 300 | 1742 | 1647 | 1646 | 1646 |
| 2 | [1,1000] | 250 | 2002 | 1975 | 1975 | 1975 |
| 3 | [1,1000] | 200 | 2538 | 2469 | 2468 | 2468 |
| 4 | [1,1000] | 150 | 3331 | 3291 | 3291 | 3291 |
| 5 | [1,1000] | 100 | 4963 | 4936 | 4936 | 4936 |
| 6 | [1,1000] | 50 | 9883 | 9871 | 9871 | 9871 |
| 7 | [200,1000] | 300 | 2084 | 2013 | 1977 | 1977 |
| 8 | [200,1000] | 250 | 2387 | 2398 | 2373 | 2372 |
| 9 | [200,1000] | 200 | 3028 | 3003 | 2965 | 2965 |
| 10 | [200,1000] | 150 | 4072 | 3989 | 3954 | 3954 |
| 11 | [200,1000] | 100 | 5957 | 5965 | 5930 | 5930 |
| 12 | [200,1000] | 50 | 11872 | 11906 | 11860 | 11860 |
| 13 | [500,1000] | 300 | 2806 | 2612 | 2501 | 2499 |
| 14 | [500,1000] | 250 | 3018 | 3177 | 2999 | 2999 |
| 15 | [500,1000] | 200 | 3793 | 3907 | 3749 | 3749 |
| 16 | [500,1000] | 150 | 5176 | 5180 | 4999 | 4999 |
| 17 | [500,1000] | 100 | 7507 | 7639 | 7498 | 7498 |
| 18 | [500,1000] | 50 | 15002 | 15167 | 14995 | 14995 |
| 19 | [800,1000] | 300 | 3518 | 3440 | 3352 | 3352 |
| 20 | [800,1000] | 250 | 3600 | 3800 | 3597 | 3597 |
| 21 | [800,1000] | 200 | 4513 | 4800 | 4497 | 4496 |
| 22 | [800,1000] | 150 | 6271 | 6420 | 6081 | 6081 |
| 23 | [800,1000] | 100 | 8995 | 9374 | 8992 | 8992 |
| 24 | [800,1000] | 50 | 17986 | 18382 | 17984 | 17984 |

Table 8 provides execution times of the Swapping Heuristic for various numbers of jobs and numbers of machines. Objects were selected at uniform random in the given range. The lower bounds are determined using Equation 19. Since the lower bounds may be less than the optimal answers, they may not be achievable.

**Equation 19**

$$\text{Lower Bound} = \left\lceil \frac{\sum_{i=1}^{n} p_i}{m} \right\rceil$$

**Table 8: Results, Minimum Makespan Problem, Uniform Distribution**

| Number of Jobs | Job Processing Time Range | Number of Machines | Longest Job First Makespan | Swapping Heuristic Makespan | Swapping Heuristic Time[1] | Lower Bound Makespan |
|---|---|---|---|---|---|---|
| 2000 | [2000,10000] | 500 | 23588 | 23474 | 109.74 | 23473 |
| 2000 | [2000,10000] | 400 | 29878 | 29341 | 214.51 | 29341 |
| 2000 | [2000,10000] | 300 | 40164 | 39121 | 307.88 | 39121 |
| 2000 | [2000,10000] | 200 | 58781 | 58681 | 86.58 | 58681 |
| 2000 | [2000,10000] | 100 | 117439 | 117361 | 29.73 | 117361 |
| 2000 | [2000,10000] | 50 | 234770 | 234722 | 6.47 | 234722 |
| 4000 | [2000,10000] | 500 | 47295 | 47256 | 184.42 | 47256 |
| 4000 | [2000,10000] | 400 | 59145 | 59069 | 426.78 | 59069 |
| 4000 | [2000,10000] | 300 | 80031 | 78759 | 273.30 | 78759 |
| 4000 | [2000,10000] | 200 | 118191 | 118138 | 90.87 | 118138 |
| 4000 | [2000,10000] | 100 | 236336 | 236276 | 28.60 | 236276 |
| 4000 | [2000,10000] | 50 | 472560 | 472552 | 10.11 | 472552 |

[1] Sparc 10 CPU seconds

Table 9 compares the performance of the SH relative to LJF and Multifit on instances with discrete distributions. These instances are created by randomly selecting ten processing times in the range [500,1000] and making 100 jobs with each of these processing times. The lower bounds are determined using Equation 20. Since the lower bounds may be less than the optimal answers, they may not be achievable.

**Equation 20**

$$\text{Lower Bound} = \left\lceil \frac{\sum_{i=1}^{n} p_i}{m} \right\rceil$$

**Table 9: Results, Minimum Makespan Problem, Discrete Distribution**

| | | | Makespan | | | |
|---|---|---|---|---|---|---|
| Number of Jobs | Job Processing Time Range | Number of Machines | Multifit | LJF | Swapping Heuristic | Lower Bound |
| 1000 | [500,1000] | 300 | 2349 | 2531 | 2262 | 2210 |
| 1000 | [500,1000] | 200 | 3542 | 3319 | 3318 | 3315 |
| 1000 | [500,1000] | 300 | 2644 | 2694 | 2468 | 2391 |
| 1000 | [500,1000] | 200 | 3717 | 3599 | 3588 | 3586 |

## 7.3.2 Comparison to Simulated Annealing

Table 10 and Table 11 compare the Swapping Heuristic to Simulated Annealing. The tables list the values of the objective function obtained with LJF, Simulated Annealing, and the SH. The SH is run until it reaches the maximum number of tries rather than stopping when the lower bound of the makespan is reached. This is required to optimize their cost function rather than finding only the minimum makespan. The SH is not altered in any other way. [Rao and Iyengar, 94] do not provide the execution times required to obtain their results, but kindly furnished the data files they used. For all runs except one, the SH produces an arrangement with an objective function value less than or equal to that of the simulated annealing.

**Table 10: Results, Minimum Makespan Problem, Uniform Distribution**

| Number of Machines | Longest Job First Objective Function Value | Simulated Annealing Objective Function Value | Swapping Heuristic Objective Function Value |
|---|---|---|---|
| 10 | 2.4 | 2.4 | 2.4 |
| 20 | 3.2 | 3.2 | 3.2 |
| 40 | 9.6 | 9.6 | 9.6 |
| 60 | 19.7 | 13.7 | 11.7 |
| 80 | 62.8 | 14.8 | 12.8 |
| 100 | 361.0 | 29.0 | 23.0 |
| 120 | 77.9 | 25.9 | 13.9 |
| 140 | 964.1 | 36.1 | 30.2 |
| 160 | 326.4 | 34.4 | 14.4 |
| 180 | 673.9 | 47.9 | 43.9 |
| 200 | 4967.5 | 47.5 | 43.5 |
| 300 | 10654.3 | 86.3 | 74.3 |

**Table 11: Results, Minimum Makespan Problem, Normal Distribution**

| Number of Machines | Longest Job First Objective Function Value | Simulated Annealing Objective Function Value | Swapping Heuristic Objective Function Value |
|---|---|---|---|
| 10 | 100.9 | 0.9 | 0.9 |
| 20 | 211.0 | 6.9 | 5.0 |
| 40 | 1074.9 | 8.9 | 8.0 |
| 60 | 1053.2 | 19.3 | 9.0 |
| 80 | 480.3 | 22.3 | 18.5 |
| 100 | 1079.2 | 27.2 | 25.0 |
| 120 | 1484.1 | 26.1 | 10.0 |
| 140 | 4584.7 | 38.7 | 10.1 |
| 160 | 7186.1 | 40.1 | 34.7 |
| 180 | 5458.9 | 35.0 | 35.7 |
| 200 | 4967.5 | 47.5 | 38.0 |
| 300 | 14742.3 | 42.3 | 41.0 |

# 8. Conclusion

In a grouping problems, a set must be partitioned subject to problem specific constraints. Two classic examples of grouping problems are the BPP and the MMP. Since these problems are NP-Complete, finding optimal solutions for large instances is computationally infeasible. Approximation schemes provide high order polynomial time algorithms that are unreasonable for practical problems. For computationally feasible methods, the focus turns to finding practical methods that require less time.

The SH stemmed from Falkenauer's work on the HGGA. The SH is family of local search heuristics that produce near-optimal solutions for many grouping problems. The SH was originally designed for use as a new heuristic in the HGGA. The SH, however, is powerful enough to be used alone. The SH to date has been applied to the BPP and the MMP.

In the BPP, Falkenauer's objective function provides a metric to guide the GGA. A local search can also use this metric. As applied to the BPP, the SH performs swaps between groups that increase the objective function. The SH outperforms FF, FFD, BF, and BFD on a wide range of instances. On Falkenauer's difficult BPP benchmark instances, the SH produces optimal solutions in far less time than Falkenauer's HGGA.

As applied to the MMP, the SH identifies the machines scheduled for the longest amount of time and attempts to shorten their schedule by exchanging jobs with other machines. The SH produces significantly higher quality solutions than LJF and the Multifit Algorithm as tested on a wide range of instances. On a problem similar to the MMP, the SH produces higher quality results than simulated annealing as proposed by Rao.

# 9. References

[Bilchev, 95] G.A. Bilchev, *Dynamics Paradigm in State Space Search and Optimization*, thesis, New Bulgarian University, 1995.

[Coffman, Garey and Johnson, 78] E.G. Coffman, M.R. Garey and D.S. Johnson, *An Application of Bin-Packing to Multiprocessor Scheduling*, SIAM Journal of Computing, Vol. 7, No. 1, February 1978.

[Falkenauer, 95] E. Falkenauer, *Solving Equal Piles with the Grouping Genetic Algorithm*, Proceeding of the Sixth International Conference on Genetic Algorithms, 1995.

[Falkenauer, 94A] E. Falkenauer, *A Hybrid Grouping Genetic Algorithm for Bin Packing*, Technical Report R0108, 1994.

[Falkenauer, 94B] E. Falkenauer, *A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems*, Evolutionary Computation, Vol 2, pp123-144, 1994.

[Falkenauer, 92] E. Falkenauer, *A Genetic Algorithm for Bin Packing and Line Balancing*, Proceedings of the 1992 IEEE International Conference on Robotics and Automation.

[Garey and Johnson, 79] M.R. Garey and D.S. Johnson, *Computers and Intractability*,.W.H. Freeman and Company, 1979.

[Goldberg 89] D.E.Goldberg, *Genetic Algorithms In Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company Inc, 1989.

[Graham, 69] R.L. Graham, *Bounds of Multiprocessing Timing Anomalies*, SIAM Journal of Applied Math, Vol 17, pp263-269, 1969.

[Hochbaum and Shmoys, 87] D.S. Hochbaum and D.B. Shmoys, *Using Dual Approximation Algorithms for Scheduling Problems*, Jomal of the Association for Computing Machinery, Vol. 34, No. 1, January 1987, pp. 144-162.

[Johnson, 82] D.S. Johnson, *The NP Complete Column, an Ongoing Guide*, Journal of Algorithms, 3, 1982, pp 288-300.

[Kao and Lin, 92] C.Y. Kao and F.T. Lin, *A Stochastic Approach for the One-Dimensional Bin-Packing Problems*, 1992 IEEE International Conference on Systems, Man, and Cybernetics V2, pp1545-51, 1992.

[Karmakar and Karp, 82] M. Karmakar and R.M. Karp, *An Efficient Approximation Scheme for the One-Dimensional BPP*, Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, 1982, pp 312-320.

[Martello and Toth, 89] S. Martello and P. Toth, *Lower Bounds and Reduction Procedures for the Bin Packing Problem*, Discrete Applied Mathematics 28 (1990) 59-70.

[Rao and Iyengar, 94] R.L. Rao and S.S. Iyengar, *Bin-Packing by Simulated Annealing*, Computers Mathematics Applications Vol. 27, No. 5, pp. 71-82, 1994.

[Vega and Lueker, 81] W. F. de la Vega C.S. Leuker, *Bin Packing can be solved within $1+\varepsilon$ in Linear Time*, Combinatorica 1 ,1981, pp 349-355.