Reduction of Printed Circuit Card Placement Time Through the Implementation of Panelization

by

John T. Tester

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
towards fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Industrial and Systems Engineering

William G. Sullivan, Chair
Kimberly P. Ellis
Subhash A. Sarin
John P. Shewchuk
Dennis W. Sweeney

Blacksburg, Virginia

Keywords: Printed Circuit Card, Component Placement, Electronic Assembly, Manufacturing

Reduction of Printed Circuit Card Placement Time Through the Implementation of Panelization

John T. Tester

(ABSTRACT)

Decreasing the cycle time of panels in the printed circuit card manufacturing process has been a significant research topic over the past decade. The research objective in such literature has been to reduce the placement machine cycle times by finding the optimal placement sequences and component-feeder allocation for a given, *fixed*, panel component layout for a given machine type. Until now, no research has been found which allows the alteration of the panel configuration itself, when panelization is a part of that electronic panel design. This research will be the first effort to incorporate panelization into the cycle time reduction field. The PCB circuit design is *not* to be altered; rather, the panel design (i.e., the arrangement of the PCB in the panel) is altered to reduce the panel assembly time. Component placement problem models are developed for three types of machines: The automated insertion machine (AIM), the pick-and-place (PAPM) machine, and the rotary turret head machine (RTHM). Two solution procedures are developed which are based upon a genetic algorithm (GA) approach. One procedure simultaneously produces solutions for the best panel design and component placement sequence. The other procedure first selects a best panel design based upon an estimation of its worth to the minimization problem. Then that procedure uses a more traditional GA to solve for the component placement and component type allocation problem for that panel design. Experiments were conducted to discover situations where the consideration of panelization can make a significant difference in panel assembly times. It was shown that the PAPM scenario benefits most from panelization and the RTHM the least, though all three machine types show improvements under certain conditions established in the experiments.

Dedication

My sincerest thanks go to Dr. William Sullivan for being my advisor and allowing me the latitude to pursue my research interests. I appreciate the contributions of the other committee members: Drs. Kimberly Ellis, Subhash Sarin, John Shewchuk and Dennis Sweeney. Their suggestions and comments helped improve the quality of this research.

My dedication of this work is to two people most dear to me in this world: My mother and father, Patricia and Dave Tester. They supported me when I left a good career to pursue the doctorate. They've been the best parents I can imagine having; I don't know how I lucked out being their son, but I'll take that luck any day.

Thanks must go to my new college friends I've created here over the past four years, particularly to Anan Mungwattana, Flor Martinez, Adar Kalir, and Andre Ramos. What little social life I had was spent chatting with these friends and they possibly saved my sanity.

And a special thanks to my newest best friend, though I have known him since I was born: My brother Dale. We didn't know each other very well until I moved back to the Southeast four years ago to enroll at Virginia Tech. Since that time, we have learned that blood is thicker than water. The time spent here for the pursuit of the PhD was worth it for that reason alone.

Lastly, I thank God for having the good fortune to be born and raised in the United States of America. No other land has such boundless opportunities for sucess, combined with such beautiful scenery and the environment for freedom of the human spirit.

Contents

1 Introduction **1**

1.1 Panelization 2

1.2 Research Problem Statement and Limitations 8

1.2.1 Problem Statement 8

1.2.2 Assumptions and Limitations 9

1.3 Research Objective and Purpose 11

1.3.1 Importance of Research 11

1.3.2 Research Hypothesis 12

1.3.3 Scope of the Research 13

1.4 Outline of Research Document 14

2 Literature Review **15**

2.1 PCCA Panelization and Industry 15

2.2 Cutting Stock/Assortment Problems 17

2.3 Component Placement 27

2.3.1 Common Traits in the Machine Types 44

2.4 Genetic Algorithms 46
2.4.1 Solution Representation 47
2.4.2 Selection Operation 48
2.4.3 Genetic Operators 50
2.4.4 Initialization, Termination, and Evaluation 52
2.4.5 GA Use in Cutting Stock and Component Placement Research 53

3 Approach to Minimization of Time for Component Placement 56

3.1 Panel Pattern Generation 57
3.2 Transformation of Rotated PCB Within Panels 62
3.2.1 Nomenclature for Addressing PCB Rotations 68
3.3 Objective Functions for Component Placement with Panelization 69
3.3.1 Automatic Insertion Machine 69
3.3.2 Pick and Place Machine 70
3.3.3 Rotary Turret Head Machine 72
3.4 The Genetic Algorithm Method 76
3.4.1 Genetic Algorithm Procedure 77
3.4.2 Solution Representation 79
3.4.3 Fitness Functions 81
3.4.4 Selection Operation 81
3.4.5 Genetic Operators 81
3.4.6 Termination Criteria 84

3.4.7 Initialization 84
3.5 The Estimator Functions 85
3.5.1 AIM Estimator Function 87
3.5.2 PAPM Estimator Function 87
3.5.3 RTHM Estimator Function 89

4 Experimental Design **92**

4.1 Justification of Experimental Parameters 93
4.2 Computer Programs and Computational Issues 98
4.2.1 GA Implementation in Distance-related Problems 99
4.2.2 Output Nomenclature 100
4.3 General Experimental Design 101
4.4 Experiment 1 106
4.5 Experiment 2 109
4.6 Experiment 3 109
4.7 Experiment 4 115
4.8 Experiment 5 117

5 Experimental Results and Discussion **120**

5.1 Experiment 1 Results 120
5.1.1 AIM 120
5.1.2 PAPM 122

5.1.3 RTHM 124
5.2 Experiment 2 Results 125
5.2.1 AIM 127
5.2.2 PAPM 130
5.2.3 RTHM 133
5.3 Experiment 3 Results 154
5.3.1 PAPM 154
5.3.2 RTHM 157
5.4 Experiment 4 Results 165
5.4.1 PAPM 165
5.4.2 RTHM 166
5.5 Experiment 5 Results 169
5.5.1 AIM 172
5.5.2 PAPM 173
5.5.3 RTHM 174
5.6 General Observations 175

6 Conclusions and Future Research 182

6.1 Summary 182
6.1.1 Objectives addressed 183
6.1.2 Results 185
6.2 Contributions 186

6.2.1 General Rules for Application 187
6.3 Future Research Opportunities 188
6.3.1 Characterization of PCB component layouts 188
6.3.2 Expansion into Entire Assembly Line 188
6.3.3 Specific Machine Types 189

Bibliography **191**

A `panelizer.C` **201**

A.1 Command arguments for panelizer.C 201
A.1.1 Batch submission examples 203
A.2 Acknowledgments and copyrights 204

B `pmaker.C` **268**

C `finder.C` **294**

D `Explanation of errors in Leu et. al. [48].` **318**

E Solution Data for Experiments 2 and 3 **322**

F `Pattern alternatives for 8 identical PCB and PAPM.` **338**

G Solution Data for Experiment 5 **341**

List of Figures

1.1 Schematic example of PCB with components. 2

1.2 Panelization and separation. 4

1.3 Manufacturing process flow for Type I, Type 2 and Type 3 PCB designs [43]. 5

2.1 Types of cuts in cutting stock research [67]. 18

2.2 Rectangular and square pieces in an assortment problem. 23

2.3 Cutting pattern made with guillotine cuts (adapted from [12]). 24

2.4 Automatic Insertion Machine (AIM) [48]. 28

2.5 Generic panel (no panelization distinction). 30

2.6 Pick-and-place machine (PAPM). 31

2.7 Required and nonrequired arcs for a PAPM [3]. 32

2.8 One head, rotational robot arm inserter. 33

2.9 Dual head placement machine (adapted from [1]. 34

2.10 Rotary Turret Head (RTHM) machine schematic. 37

2.11 Typewriter heuristic example. 40

2.12 S-shape heuristic example. 40

2.13 FPP model (adapted from [71]). 43

2.14 DPP model (adapted from [71]). 43

2.15 Layout of two-concurrent-robot assembly workstation (adapted from [50]). . 44

2.16 Layout of two-robot sequential assembly workstation (adapted from [50]). . . 45

2.17 Representation of pattern position and orientation [38]. 54

2.18 Example of a 21 component, 3 component type chromosome [48, 49, 83]. . . 55

3.1 Mirrored PCB shapes result in different component locations for entire panel. 59

3.2 Local PCB Component Location at $O_{fk} = 0$. 64

3.3 Panel Schematic. 65

3.4 Local PCB Component Location for $O_{fk} = 1$. 67

3.5 Rotary turret head machine (RTHM) schematic. 73

3.6 Panel example. 77

3.7 Simple example of two PCB in a panel design. 86

3.8 Graphical illustration of EF(AIM), EF(PAPM) and E(RTHM). 88

4.1 Histogram of panel component populations in literature. 94

4.2 Histogram of panel component type populations in literature. 95

4.3 Histogram of panel dimensions in literature. 95

4.4 Pattern alternatives for the AIM type, 8-PCB experiments. 97

4.5 Experiment 2, Cases A, B and C for 2-PCB panels. 111

4.6 Experiment 2, Cases A, B and C for 4-PCB panels. 112

4.7 Experiment 2, Cases A, B and C for 8-PCB panels. 113

4.8 Experiment 3 for 2, 4 and 8-PCB panels. 114

4.9 Experiment 4, component types 115

4.10 An industrial example of a panelized PCB layout (Experiment 4). 116

5.1 Best solution sequence for AIM problem from Leu et. al. [48]. 121

5.2 Best solution for the AIM experiment 1 problem using `panelizer` 121

5.3 Comparison of `panelizer` convergence against Leu et. al. [48] for AIM experiment 1. 122

5.4 Best solution sequence for PAPM experiment 1 from `panelizer`. 123

5.5 Comparison of `panelizer` convergence against Leu et. al. [48] for PAPM experiment 1. 123

5.6 Best solution sequence for RTHM experiment 1 from `panelizer`. 124

5.7 Solution trend for RTHM experiment 1 case. 125

5.8 Best and Worst Estimator results for 2-PCB, AIM experiment 2. 136

5.9 Solutions for AIM experiment 2 for 2 PCB. 137

5.10 Best and Worst Estimator results for 4-PCB, AIM experiment 2. 138

5.11 Solutions for AIM experiment 2 for 4-PCB. 139

5.12 Best and Worst Estimator results for 8-PCB, AIM experiment 2. 140

5.13 Solutions for AIM experiment 2 for 8-PCB. 141

5.14 Best and Worst Estimator results for 2-PCB, PAPM experiment 2. 142

5.15 Solutions for PAPM experiment 2 for 2-PCB. 143

5.16 Best and Worst Estimator results for 4-PCB, PAPM experiment 2. 144

5.17 Solutions for PAPM experiment 2 for 4-PCB. 145

5.18 Best and Worst Estimator results for 8-PCB, PAPM experiment 2. 146

5.19 Solutions for PAPM experiment 2 for 8-PCB. 147

5.20 Best and Worst Estimator results for 2-PCB, RTHM experiment 2. 148

5.21 Solutions for RTHM experiment 2 for 2-PCB. 149

5.22 Best and Worst Estimator results for 4-PCB, RTHM experiment 2. 150

5.23 Solutions for RTHM experiment 2 for 4-PCB. 151

5.24 Best and Worst Estimator results for 8-PCB, RTHM experiment 2. 152

5.25 Solutions for RTHM experiment 2 for 8-PCB. 153

5.26 Best and Worst Estimator results for 2-PCB, PAPM experiment 3. 159

5.27 Solutions for PAPM experiment 3 for 2-PCB. 159

5.28 Best and Worst Estimator results for 4-PCB, PAPM experiment 3. 160

5.29 Solutions for PAPM experiment 3 for 4-PCB. 160

5.30 Best and Worst Estimator results for 8-PCB, PAPM experiment 3. 161

5.31 Solutions for PAPM experiment 3 for 8-PCB. 161

5.32 Best and Worst Estimator results for 2-PCB, RTHM experiment 3. 162

5.33 Solutions for RTHM Experiment 3 for 2-PCB. 162

5.34 Best and Worst Estimator results for 4-PCB, RTHM experiment 3. 163

5.35 Solutions for RTHM Experiment 3 for 4-PCB. 163

5.36 Best and Worst Estimator results for 8-PCB, RTHM experiment 3. 164

5.37 Solutions for RTHM Experiment 3 for 8-PCB. 164

5.38 Best and Worst Estimator results for 8-PCB, PAPM experiment 4. 167

5.39 Solutions for PAPM experiment 4 for 8-PCB. 167

5.40 Best and Worst Estimator results for 8-PCB, experiment 4. 168

5.41 Solutions for RTHM experiment 4 for 8-PCB. 168

5.42 Comparison of AIM experiment 2 results over PCB population. 180

5.43 Comparison of PAPM experiment 2 results over PCB population. 180

5.44 Comparison of RTHM experiment 2 results over PCB population. 181

D.1 Spreadsheet calculations for Chebyshev solution by Leu et. al. 320

D.2 Spreadsheet calculations for Euclidean solution by Leu et. al. 321

List of Tables

2.1 Assortment problem in panelization terminology. 19

2.2 Additive pattern-building terminology[79]. 24

2.3 Four rules used in the knowledge-based approach by Yeo and Yong[84]. . . . 36

3.1 Panel pattern terminology. 58

3.2 Transformation terminology. 63

3.3 AIM formulation terminology. 70

3.4 PAPM formulation terminology. 71

3.5 RTHM formulation terminology. 74

3.6 Layout of chromosome links. 80

3.7 Genetic Operators. 82

4.1 Experiment E1: Verification. 102

4.2 Experiment E2: Eccentric component group locations. 103

4.3 Experiment E3: Component type eccentricity. 103

4.4 Experiment E4: Industry PCB/panel design example. 103

4.5 Experiment E5: Random PCB designs. 104

4.6 AIM component locations [48]. 106

4.7 PAPM component locations (X, Y) and types (CT) for verification example [48]. 107

4.8 RTHM component locations (X, Y) and types (CT) for verification example [48]. 108

5.1 Experiment 1 results. 120

5.2 Experiment 2 summary of results. 126

5.3 Experiment 3 summary of results. 155

5.4 Experiment 4 summary of results. 165

5.5 Experiment 4 summary of results (LPI). 165

5.6 Experiment 5 summary results, AIM. 169

5.7 Experiment 5 summary results, PAPM. 170

5.8 Experiment 5 summary results, RTHM. 170

5.9 Experiment 5 summary *average* results with Confidence Intervals (CI). . . . 171

5.10 Experiment 5 correlation between results. 172

5.11 Distance from PCB geometric center to center of component distribution. . . 177

5.12 Experiment 2 correlation between results. 179

G.1 Experiment 5 for AIM and 2 PCB. 342

G.2 Experiment 5 for AIM and 4 PCB. 342

G.3 Experiment 5 for AIM and 8 PCB. 343

G.4 Experiment 5 for PAPM and 2 PCB. 343

G.5 Experiment 5 for PAPM and 4 PCB. 344

G.6 Experiment 5 for PAPM and 8 PCB. 344

G.7 Experiment 5 for RTHM and 2 PCB. 345

G.8 Experiment 5 for RTHM and 4 PCB. 345

G.9 Experiment 5 for RTHM and 8 PCB. 346

Glossary

1. *AIM.* Automated Insertion Machine.

2. *Center of panel component distribution* (X^{CP}, Y^{CP}). The center of distribution for all the components on the panel.

3. *Center of PCB component distribution* (x_k^{CP}, y_k^{CP}). The center of the component distribution, assuming all components have equal weight and relative to the PCB geometric center.

4. *Center of PCB component type distribution* $((x_{kj}^{CTP}, y_{kj}^{CTP})$). The center of the component distribution according to component types.

5. *Component.* An electronic device which is mounted permanently upon the PCB.

6. *Component type.* A component can be of a particular type, distinct from other components in a PCB design; components of like types can be assigned to a resource location in an assembly machine. $\overline{N}_k$ is the total component types on a PCB k and $\overline{N}$ is the total of all distinct type on the panel.

7. *Composite Chromosome.* A GA solution form which has separate, distinct links which must all be included to produce a solution using the objective function.

8. *LPI Consistency.* The similarity between TLPI and GLPI results for the same problem under consideration.

9. *Cycle time.* The time required for a PCCA machine to place all the components on the panel. In reference to the AIM and PAPM and in the context of this research, the "cycle" is directly correlated to the planar distance traveled by the placement head.

10. *Distance Matrix.* In an AIM situation, the a matrix of all possible distances between a given component and all the others. In a PAPM situation, the a matrix of all possible distances between each component and each feeder slot location.

11. *Eccentricity (of component layout).* The amount of difference between the center of the PCB component distribution and the geometric center of the PCB.

12. *EF(AIM), EF(PAPM), and EF(RTHM).* The estimate value of the machine type; relative to other panel designs for the same problem, estimates the potential the design for good (or bad) component placement time results.

13. *Estimated Best (EB).* This design is selected through the Estimator Function which yields the design producing the lowest estimator results for that experimental instance.

14. *Estimated LPI (ELPI).* Based upon the difference between the Estimated Best and Worst scores for a given panel and machine scenario. This value *does not* represent a prediction of what a traditional or global analysis score would be; rather, the ELPI is compared against other design alternatives for a relative ranking of these designs for potential best solutions. Calculated as $ELPI = \frac{EW-EB}{EW} \times 100$.

15. *Estimated Worst (EW).* This design is selected through the Estimator Function which yields the design producing the highest estimator results for that experimental instance.

16. *Estimator Function.* A function developed to estimate a particular panel design's relative potential for improved component placement times with respect to alternative panel designs under consideration.

17. *Experimental Instance.* An experiment for a particular PCB component layout, run for the given machine type and panel layout.

18. *Feeder slot.* A location in the PCCA machine where components of an assigned type can be presented for pick up by the placement head or gripping device.

19. *Genetic Algorithm (GA).* A search technique which uses stochastic methods, combined with the ranking of multiple solutions through many iterations, to arrive at a continually improving solutions.

20. *Global Approach.* Conducting a panelization approach by searching the entire panel design space simultaneously. The component placement sequence, component type allocations, panel pattern and PCB rotation set are all determined as a result of the solution.

21. *Global LPI (GLPI).* Based upon the difference between the Traditional Worst and global analysis results for a given panel and machine scenario. Calculated as either $GLPI = TW - G$ or $GLPI = \frac{TW-G}{TW} \times 100$.

22. *Guillotine cutting pattern.* A pattern of shapes within a rectangular boundary which can be formed by complete, straight-line cuts across that boundary.

23. *Largest Potential for Improvement (LPI).* The difference found between minimization algorithm results for two different panel designs. It is anticipated that such a difference indicates how much improvement in the panel cycle time is possible if the panel designer chose the best panel design with respect to choosing the worst one. difference between a best panel design score found for a given number of PCB required in a panel and the worst panel design score.

24. *Panelization.* The organization of multiple product PCB designs on a single panel. It included both the locations of the PCB within a panel boundary as well as the feasible PCB rotations within those locations.

25. *Panelization Approach.* Component placement sequencing and allocation problems accomplished with consideration of panel design alternatives.

26. *PAPM.* Pick-and-Place Machine.

27. *PCB geometric center (or PCB center).* Half the PCB width and length; this is the reference point for the component layout and the PCB orientation.

28. *Panel design.* The information which dictates the panel construction. Composed of the pattern and the PCB rotation set.

29. *Printed Circuit Board (PCB).* The fiberglass substrate upon which a circuit design is etched; the PCB circuit design determines the locations of the components on that PCB.

30. *Printed Circuit Card Assembly (PCCA).* The manufacture of printed circuit cards; in the context of this research, it is assumed that such cards are assembled via automated machinery.

31. *Pattern.* A set of PCB shapes which form the panel dimensions. These shapes are defined as rectilinear positions of the PCB geometric centers as well as the PCB orientation necessary for the PCB to fit within the pattern.

32. *PCB shape.* The dimension boundaries of a PCB. Considered as either a rectangle or a square.

33. *PCB rotation set.* The PCB rotation, relative to the PCB orientation, within a specific pattern alternative.

34. *PCB orientation.* The rotation of the PCB, about its geometric center, within a pattern, relative to its PCB component layout

35. *PCB component layout (PCB design).* The locations and component types (where applicable) of the PCB.

36. *RTHM.* Rotary Turret Head Machine. Also known in industry as a "chip-shooter."

37. *Traditional Approach.* Component placement sequencing and allocation problems accomplished without considering the panel design alternatives. This method has been the means by which such problems have been addressed to this time.

38. *Traditional Best (TB).* The minimum placement time (or distance) produced from a predesignated Estimated Best panel design through the use of a traditional GA analysis.

39. *Traditional GA.* A GA conducted where only the component placment sequence and component type allocation (when appropriate) are determined by the solution. The panel design (pattern and PCB rotation set) is fixed before the GA is begun.

40. *Traditional LPI (TLPI).* Based upon the difference between the Traditional Best and Worst scores for a given panel and machine scenario. Calculated as either $TLPI = TW - TB$ or $TLPI = \frac{TW-TB}{TW} \times 100$.

41. *Traditional Worst (TW).* The minimum placement time (or distance) produced from a predesignated Estimated Worst panel design through the use of a traditional GA analysis.

42. *Two-Stage Analysis.* Conducting a panelization approach in two stages: 1) Estimating the best and worst panel design alternatives and then 2) conducting a traditional approach analysis on those alternatives.

Chapter 1

Introduction

Research in printed circuit card assembly (PCCA) has been of interest for the last several years, from an industrial engineering point of view. The business of PCCA manufacturing continues to grow; over two billion dollars worth of printed circuit board (PCB) sales were accomplished by the top ten North American manufacturers in 1996 [57]. In 1992, a workshop was conducted to discuss PCCA research; a question from the event was posed by Wilhelm and Fowler [82]: "How can we better understand what is unique about circuit card assembly?" Based upon work experience and research, it was discovered that printed circuit board panelization was one of those unique aspects of printed circuit card assembly which has apparently been largely ignored in academic literature.

A PCB design is the etched pattern produced in a laminated panel; components are placed upon that panel, based upon the circuit design. This design represents a single product, component layout, as illustrated in Figure 1.1. In the manufacture of smaller, often hand-held devices (such as cellular phones or calculators), the boards are of such small dimensions that multiple boards are etched on a single, larger panel (Figure 1.2). This method gives rise to a manufacturing panel which is, in itself, a conceptual "batch" of like products which are assembled together in the printed circuit card assembly line. The organization of multiple product PCB designs on a single panel is termed panelization [41].

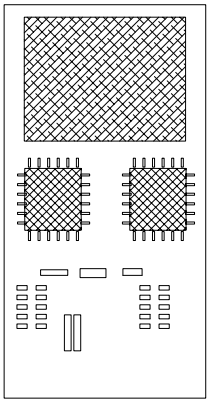


Figure 1.1: Schematic example of PCB with components.

At first, this problem may seem to be only involved with nesting a smaller geometry into a larger panel dimension. However, from a manufacturing point of view, the panel takes on integer sets of alternative component locations. The manufacturing process requires that each component be accurately placed upon the panel and secured. With potentially hundreds of components on a single panel, the most efficient means of placing the components is via automated machinery.

Panelization segregates product characteristics into integer sets of alternative panel designs. This method of designing leads to a question associated with its manufacture: How do these designs impact the manufacturing process? By examining the characteristics of panelization, one will discover that a significant area of PCCA manufacturing — electronic component placement — is highly influenced by such panelization.

1.1 Panelization

In order to illustrate how component placement is affected by panelization, discussions of PCB design and PCCA production are in order.

A PCB may have several characteristics associated with its design and construction. The substrate is the composite, usually fiberglass, sheet which is fabricated with the circuit

trace designs embedded within it. The traces are conductive paths within the board; these paths usually emerge from within the board to the board surface, the end of which are conductive pads. These pads geographically locate the placement of electronic components upon the substrate surface. Components which are physically located in such a fashion are designated as surface mount devices (SMD). The other, common means by which PCB components may be integrated into the circuitry is called insertion-pin-through-hole (IPTH). These devices have leads which are pushed through holes within the board and soldered in place. The holes are locations where a trace may end for the purpose of such a component connection. SMD components are nearly always mounted with surface mount technology (SMT) machines; these machines are very large, very fast (in terms of moving components from bulk locations to PCB locations) and very expensive. IPTH components may or may not be placed on the board with an automated machine; such a machine is often termed an automated insertion machine (AIM). More and more in today's PCCA industry, a concerted effort is made by the PCB designer to convert any IPTH component to an SMD equivalent, because the SMD equivalent can be more quickly mounted than a IPTH component. Additionally, SMD components are becoming economically competitive with respect to IPTH components in tokays market [43]. The PCB can have a combination of SMD and IPTH, or may be composed exclusively of one or the other type. A panel is the unit product which is processed by the manufacturing line. As mentioned earlier, this panel can be composed of multiples of the PCB design, all produced from a single, physically unified board. It should be noted, however, that due to possible demand differences between models, the different PCB models are usually panelized separately; i.e., each panel contains only multiples of one PCB type. Though all PCB are shown in Figure 1.2 with the same orientation, the panel may contain PCB with different orientations. The PCB are almost always arranged in some orthogonal combination and at least one manufacturer explicitly favors a guillotine cut pattern [41]. Whatever pattern is chosen for a panel, that pattern is the only pattern used for the entire demand of the product(s) on that panel. This concept of pattern selection is very different from that of "cutting stock" problems, where each panel pattern is allowed to

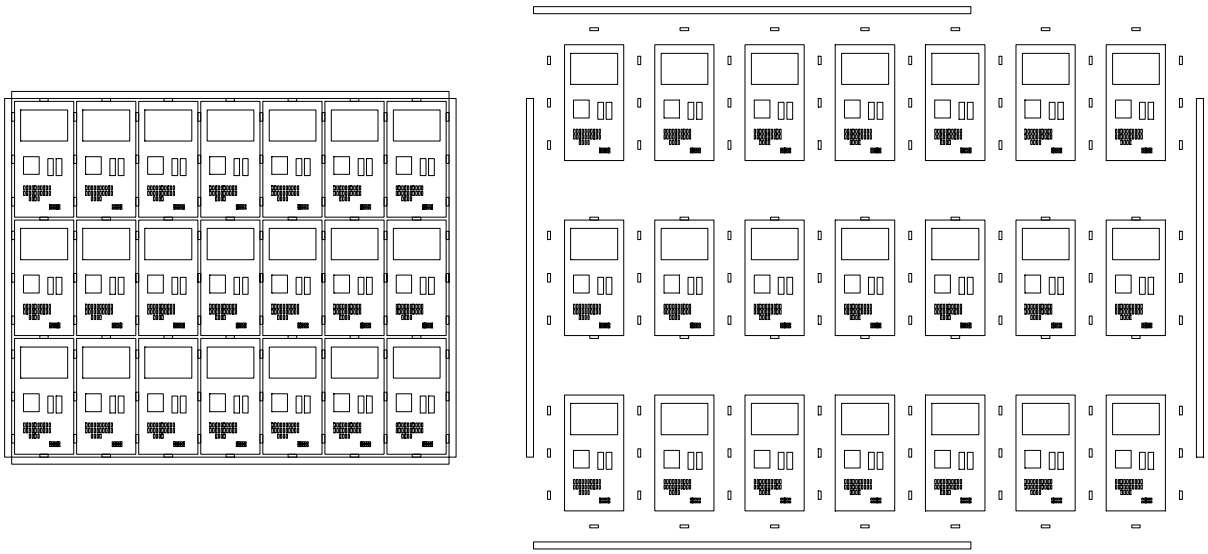


Figure 1.2: Panelization and separation.

be different for the entire demand. Cutting patterns and the cutting stock problem will be discussed further in Chapter 2.

The PCB (and panel) are classified in terms of how they are processed by the Surface Mount Council [70, 43] as Type 1, Type 2 or Type 3. A Type 1 assembly has components mounted on only one side of the substrate. A Type 2 assembly has components mounted on both sides of the substrate. An additional type, Type 3, can be described as having both IPTH and SMD components mounted on both sides of the PCB. Process flow examples of each type are illustrated in Figure 1.3. When discussing component placement problems in such a manufacturing environment, the researcher is only concerned with addressing the individual blocks entitled, "Place SMD" or " Insert and Cinch IPTH." Though this research addresses the placement machines, one should note that other processes may be affected by panelization to varying degrees as well [76].

The PCB design process generally does not explicitly address panelization as a means by which process times can be improved. Panelization is usually discussed in the industry literature as an issue associated with component or trace clearances [66, 13, 65]. "Design

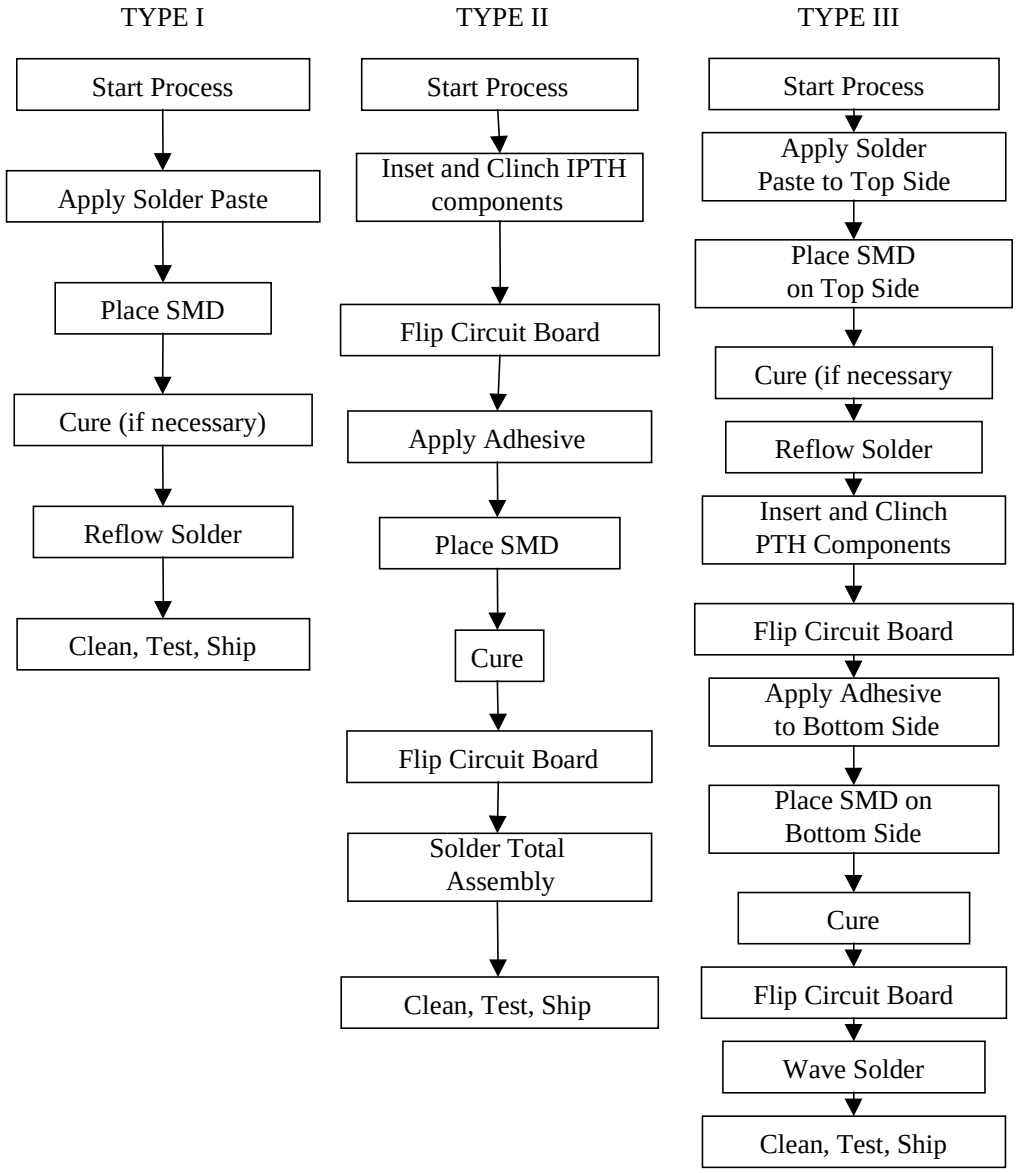


Figure 1.3: Manufacturing process flow for Type I, Type 2 and Type 3 PCB designs [43].

for manufactureability" is also cited as a category in which panelization is considered, but it is described in terms of component placement, alignment and no-place zone restrictions [3, 65, 51]. Another rationale for panelization design is the desire to reduce the waste in the panel from the PCB fabricator or to reduce the number of panels necessary for the demand [68, 39].

In all of the publications related to panelization and process time improvements(from Chapter 2), there is an assumption that an increase in the number of PCB in a panel will increase the number of individual component placements. This increase will reduce the number of panel setups for a given demand, since the more PCB allowed per panel will reduce the total number of panels processed. However, panels with an increased PCB population will increase the number of component placements and therefore increase the individual panel cycle times. The assumption is that the added processing time per panel, for all panels, is offset by the time saved due to the fewer panel setups required at each station for the overall demand.

When discussing processing time minimization for PCCA, considerable emphasis in the research literature is placed upon the total panel placement time within a component placement machine. The actual cycle time within a placement machine consists of loading, clamping (fixturing), placement and unloading the product [78]. Placement time is directly related to the number and types of the components on a panel, but the other activities' durations are typically fairly constant values. Thus, the contribution of placement time to the overall cycle time of a particular machine is increasingly important as the number of components per panel increases. A simple example by van de Vall illustrates this tendency: A panel with seven seconds of overhead time (the loading, fixturing and unloading times) may require 30 seconds of placement time for a 100 component board; for the same machine, a 500 component board may require 157 seconds of placement time [78]. Thus, component placement involves 81 percent of the cycle time for the less populated product, but nearly 96 percent of the time for more densely populated products.

This thinking is the rationale behind continuing research into the minimization of the ideal placement times for these machines. A new solution method which produces better results for the placement times will have a great impact in the cycle times for real-world applications.

Placement time reduction techniques currently vary from different types of machines. Essentially, a placement machine may have any of the following subsystems: A placement head, a single or multiple gripper mounted upon that head, a panel mounting table, and a component-supply feeder. Any or all of these subsystems can have motion capability, relative to each other, in multiple degrees of freedom (DOF). The different combinations of these subsystems and their relative DOF result in different problem formulations required for describing the component placement time. Thus, as will be shown in Chapter 2, the techniques for reducing the placement times will be unique for each machine. A representative set of machine types can be used to show how the interaction of the subsystems introduced above can affect the problem formulations and solution techniques. Some researchers state that there are three basic types of these machines: The robotic insertion machine, the cartesian or gantry type machine, and the rotary turret head or "high speed chip shooter" [60]. Similar categorization is made by other authors; for this research, these machines will be discussed as the automatic insertion machine (AIM), the pick-and-place machine (PAPM), and the rotary turret head machine (RTHM) [48, 83, 49]. As will be shown in Chapter 2, these machines can have a variety of configurations, even within these loose categories.

The formulation of component placement problems is combinatorially complex and difficult to solve optimally. Traditionally, mathematical algorithms have been developed for specific placement machine types to give good, though not optimal, solutions. In recent years, genetic algorithms (GA) have been applied to nonpolynomial (NP) hard problems such as those found in component placement problem formulations. When addressing the placement problems, the GA have the advantage over machine-specific algorithms in that the GA can be applied in similar problem constructions for a wide variety of machine types [48]. For this reason, GA will be used to solve the combined component placement/panelization

problems presented in this research.

1.2 Research Problem Statement and Limitations

Decreasing the placement cycle time of panels in the PCCA manufacturing process has been a significant research topic over the past decade as will be shown in Section 2.3. The research objective has been to reduce the placement times by finding the optimal placement sequences and component/feeder allocation for a given, *fixed*, panel component layout for a given machine type. Interestingly, no research has been undertaken to allow the alteration of the panel design itself, when panelization is a part of that panel design. This research will be the first effort to incorporate panelization into the literature dealing with cycle time reduction.

1.2.1 Problem Statement

The problem is to reduce the placement time of a panel through the examination of panel design options, when the panel is designated for component placement on a particular type of PCCA machine. The following design parameters are used to reduce the placement time:

1. Panel pattern alternative.
2. PCB rotations.
3. Placement sequence of components.
4. Feeder slot assignment of component types (as necessary).

Parameter 1 refers to a specific panel layout of all the required, separate PCB designs in the panel. Parameter 2 represents the specific PCB angular rotations for which a PCB can feasibly fit within a specific panel pattern alternative. These parameters are not typically

addressed in component placement literature and will be introduced in Section 3.1. The last two parameters are commonly used in component placement research to reduce the cycle time of the placement machines; these topics are addressed in Section 2.3.

1.2.2 Assumptions and Limitations

There are assumptions and limitations associated with the problem. A PCB circuit design is developed by the circuit designer, engineer, panel fabricator or a combination of such personnel with the additional input of a manufacturing engineer. As suggested earlier in Section 1.1, the manufacturing engineer usually provides input as to component/trace clearances, thermal considerations of solder flow, and so on. Once the m PCB_k are designed, these products are then arranged in a pattern for the panel design. This activity is involved with the PCB substrate fabrication and delivery of the unpopulated panels to the PCCA facility. The procurement of these panels is accomplished by a procurement agent only when both the m PCB_k and panel layouts are established. The panel fabricator then manufactures and delivers the panels to the PCCA site. The manufacturing engineer or technician develops the component placement sequence and feeder allocations for the single machine upon which this panel is to be populated.

There are several important limitations in the scenario described above:

1. *The problem statement takes only one placement machine under consideration at a time.* Many PCCA facilities will place components on the same panel by dividing up the placement workload upon several machines in series. This practice reduces the total placement time for the panel in any one station, and thus reduces the cycle time of the panel in the line. It also allows the facility layout to have several placement machine types in a line to handle different types of component types or placement situations. The reason that only a single machine is considered for each problem is due to the fact that the introduction of panelization parameters into the component

placement time problem is a new concept; hence, a simplified approach is required for the initial introduction of this concept into the research literature.

2. *The allocation of the panel components, designated for placement in the specific machine under consideration, has already been specified by the time of panel design.* Related to the previous assumption, this item highlights the practice of process planners to dynamically determine which panels are to be produced on a variety of available PCCA lines for any anticipated time period. This practice allows process planners the flexibility to handle the daily or weekly changes in priorities associated with the different electronic products as well as those associated with machine downtimes or utilization. PCCA process planning for manufacturing facilities is not addressed in the present research, though such complex planning activities is part of an on-going research area [52, 53, 47].

There are several definitions which will help describe the assumptions associated with the problem statement:

1. Panelization is considered as the feasible locations of PCB within a panel boundary. It will also include the feasible rotations of those PCB within fixed positions inside that panel boundary. The panel design is defined as the combination of these two factors.
2. There are m PCB_k in the panel, where k identifies the individual PCB.
3. There are n_k components for each PCB_k.
4. There are $\overline{N}_k$ component types for each PCB_k.

These are assumptions associated with the problem statement:

1. The panel is rectangular.
2. Each PCB_k is rectangular.

3. The PCB_k may be identical or of different shapes and circuit design (component locations).
4. A guillotine cutting pattern is assumed.
5. All m PCB_k can fit within the panel dimensions without overlap.
6. No waste area is allowed within the panel.
7. All placement head and table x–y motions are at a uniform, constant velocity (where applicable).
8. The placement head and/or table motions are described via an Euclidean metric.
9. Vertical head motions are ignored.
10. Each component occupies only one feeder slot of a total of S number of slots. Therefore, $\overline{N}_k = S$.

Detailed discussions of these assumptions and associated variables are covered in Chapter 3.

1.3 Research Objective and Purpose

The research in this dissertation proposes a novel approach for reducing component placement times. The PCB circuit design or the placement machine design are not to be altered; rather, the panel design decisions are made to reduce the placement times and thereby decrease the machine cycle time.

1.3.1 Importance of Research

The panelization of PCCA products is traditionally considered as a feature late in PCB product development. Global competition requires new ways of thinking within the design

process, such that the product can be produced less expensively. In the PCCA manufacturing industry, one way in which a product may be produced less expensively is by reducing the cycle time at the processing workstations; this reduction is often accomplished at the automated placement machines. If the consideration of panel design can be shown to reduce the placement machine cycle times, without affecting the PCB circuit design, then the producers that incorporate this concept will have gained an advantage over their competitors. Moreover, since no PCB circuit design will have been altered as a part of this process, addressing panelization should have a small to no impact in terms of additional cost needed to implement this process.

This research has interest to the academic community in that the problems addressed in this research are reducible to problems which are considered as NP-hard. Such problems are not solvable in polynomial time; as such, the minimization problems associated with panelization are also at least as difficult to solve. By developing search heuristics to achieve solutions to these problems this research can provide insight to similar mathematical problems.

1.3.2 Research Hypothesis

- *The Research Question*: What the research intends to answer, and how it will expand the academic body of knowledge. *Can parameters of panelization be varied to produce a lower component cycle time than if that panel were not analyzed in such a fashion?*
- *The Research Purpose*: What is the overriding reason for doing this research? *To introduce the concept of panelization into the PCCA design and manufacturing process, in order to improve panel placement times.*
- *The Research Objective*: What will be the results of this research, or what can be learned from this research? *This research will show, through the development and use of a specific approach, how the introduction of panelization into the traditional*

component placement analysis can improve the cycle times for some PCB/panel designs and placement machines.

1.3.3 Scope of the Research

This research will restrict itself to those machines which are most frequently addressed in the current research literature. Given that each placement machine manufacturer has its own unique machine characteristics and more machines are developed each year, this restriction will allow the research to become manageable. Also, some mathematical assumptions (such as only one feeder slot assigned per component type) will be used in accordance with current research literature. These assumptions will be defined as the problem formulations progress and are reasonable in the context of a new concept introduction (i.e., panelization), relative to the conventional research publications.

The solution method developed will use the GA technique as a means by which to arrive at good solutions. This technique is known for its ease of adaptation to combinatorially difficult problems, it does not guarantee optimal solutions.

The example problems examined in this research will use only guillotine cuts. This restriction, common in cutting stock and assortment problem research, allows for the use of proven methods to generate pattern alternatives for given panel dimensions.

The panel design will exclude the occurrence of void space within the panel boundaries; that is, all of the panel area must be occupied by some PCB, and there is no PCB overlap. This restriction may be considered "trim waste" in the cutting stock literature [79].

1.4 Outline of Research Document

The rest of this document is organized as follows: Chapter 2 reviews the research literature related to this field. Because this research is a new approach to reducing PCCA cycle times which combines several different research areas, Chapter 2 is separated into sections relevant to those areas. Chapter 3 covers the problem formulation and methodology developed to solve the problem. Chapter 4 begins with verification cases using published genetic algorithm cases for component placement and allocation problems; these cases are used against those programs developed in this dissertation, in order to evaluate the impact of adding panelization factors to the programs. Chapter 4 presents the experimental design and related computational issues associated with the computational data collected as part of this research. Chapter 5 presents the results from these experiments along with observation and discussion. Lastly, Chapter 6 covers the broad implications of the research results, lists research contributions, and identifies future research possibilities related to this body of investigation.

Chapter 2

Literature Review

The survey of literature is loosely divided into areas which appear to have a major influence in defining and solving the problem. These areas are:

- PCCA Panelization and Industry
- Cutting Stock/Assortment Problems
- Component placement
- Genetic Algorithms

2.1 PCCA Panelization and Industry

Panelization has been generally discussed in Chapter 1. Nearly all panelization concepts published occur in industrial journals, conference proceedings and relevant publications are discussed for background purposes in Chapter 2.

Panelization was proposed as a method by which to improve the efficiency of several, individual, PCCA manufacturing processes [66]. Whereas a single PCB requires a certain

time for screen printing, multiples of the same PCB can be on a panel which requires essentially the same amount of screening time. Rapczynski also noted that the off-line programming of an array of PCB is simple for panelized products, since most placement machines have "step and repeat" capability, where the single PCB pattern (components and locations) are recorded in placement software; the PCB pattern is then defined relative to the first PCB pattern to give the panel's entire placement layout. He also pointed out that the maximum dimensional limit of panel size may be due to potential bowing in the substrate, during reflow or solder wave processes. A suggested rule of thumb is to limit the maximum panel size to 9" x 12" for 0.062" substrates and roughly half that area for 0.032" substrates. Another expert states that the maximum panel size is due to processing machine dimensional limits [70].

PCMCIA cards for laptops have reduced the standard thicknesses beyond that suggested by Rapczynski [66]; the substrates for these products start at 0.033" and decrease to 0.018" thick for 4-layer boards [39]. These products are so thin that carriers to support the panels are common. Even so, the limit of panelization is suggested by Jeffery to be approximately 4 x 1; that is, 4 columns of identical PCB in one row [39]. This dimension is roughly 2.5" x 8". Any greater size can result in premature depanelization during inter-station handling, and lesser patterns require "excessive real estate for rails [39]." Interestingly, Rapczynski's suggestions of 1989 seemed to stay relevant in 1995 with the advent of Jeffery's suggestions.

Very little academic research has been published concerning panelization as it relates to PCCA manufacturing, or even PCB panel design. Mornar et. al. [59] produced an exception to this observation. They investigated the problem from the PCB fabricator's viewpoint. Their objective was to find an algorithm that would minimize the number of panels the fabricator was required to produce for a given order of varied products with different shape constraints. Their heuristic algorithm worked for guillotine and non-guillotine pattern formulations.

The concept of using the panel design to reduce the manufacturing cycle time was first introduced by Tester [76]. Simple cases were analyzed via complete enumeration of possible panel designs, combined with a genetic algorithm approach for solving the component placement sequence [77]. These experiments involved only 16 components per panel and did not include component type placement elements to the problem. Though simple in nature, the results showed that for at least those simple cases, the consideration of panel design in the cycle time reduction problem had the potential to produce tangible benefits.

A brief description of these terms and others related to cutting stock and stock sizing problems will be covered in the next section.

2.2 Cutting Stock/Assortment Problems

Panelization involves placing PCB of given dimensions within the larger panel borders. Both the PCB and the panel are considered rectangular in shape. This broad concept might seem to match that of cutting stock and assortment problems, which are widely researched in operations research and industrial engineering literature. A brief review into cutting stock problems will show that panelization does not quite fit into such a category, though some of the techniques in approaching such problems may be useful.

Cutting stock problems are widely addressed in the literature; they are usually concerned with reducing the wasted area in a stock sheet or minimizing the total number of stock panels necessary to meet demand [31]. For one-dimensional problems, with a single stock size specified, this problem can be solved via knapsack techniques [23]. In cutting stock problems, one or both of the stock rectangular dimensions are usually fixed for such a demand [67]. Stock sizing problems are a related area of research; they require that the stock dimensions be determined in addition to the cutting patterns for each stock sheet [72].

Frequently used in these problem definitions are the terms guillotine, non-guillotine and non-orthogonal cuts; these cuts are illustrated in Figure 2.1 [67]. In panelization terms,

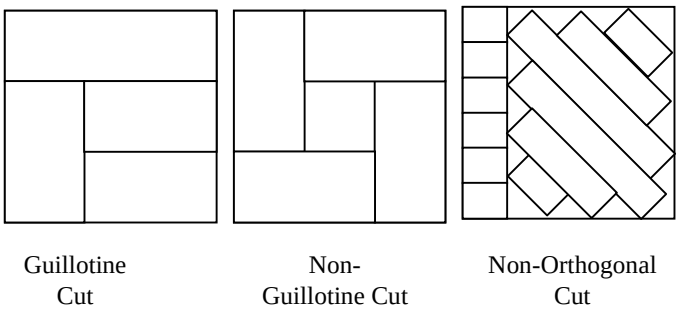


Figure 2.1: Types of cuts in cutting stock research [67].

guillotine cuts may be considered as necessary when the depanelization process uses straight-edged shears, usually as a result of scoring performed on the laminate [58]. However, though non-orthogonal panels are uncommon, panels with machine-routed patterns can take on non-guillotine cuts [41].

The stock sizing problem has also been described as an assortment problem in two dimensions [4]. Thus, the assortment problem can be considered as the most general form of these problems studied in the research literature. For illustration, the assortment problem is formulated in the PCB panelization terms of Table 2.1 [4].

Table 2.1: Assortment problem in panelization terminology.

n	Number of different possible stock types (sizes) from which to select
L_i	Length of stock rectangle type i
W_i	Width of stock rectangle type i
f_i	The fixed cost associated with the use of a panel of stock size i
k	The maximum number of different types of stock rectangles one can select for production
c_w	Cost per unit area wasted in a stock rectangle (panel)
m	Number of differently-sized, rectangular pieces from which to fit into the stock rectangles
M	A large, positive constant
y_i	$= 1$ if stock rectangle type i is used $= 0$ otherwise
l_j	Length of a type j rectangular piece
w_j	Width of a type j rectangular piece
v_j	Value associated with a rectangular piece
a_j	Lower limit on the number of rectangular pieces of type j cut $(a_j \leq 0, j = 1, \ldots, m)$
b_j	Upper limit on the number of rectangular pieces of type j cut $(b_j \leq a_j \geq 0, j = 1, \ldots, m)$
$P(i)$	The set of all cutting patterns possible for a stock rectangle of type i.
d_{jpi}	Number of pieces of type j that occur in cutting pattern $p \in P(i)$
x_{pi}	Number of times pattern $p \in P(i)$ is used for cutting a stock rectangle of type i
u_j	Number of rectangular pieces of type j cut to meet the demand of this type

$$\textit{Minimize:} \quad \sum_{i=1}^{m} \sum_{p \in P(i)} (f_i + c_w L_i W_i) x_{pi} - \sum_{j=1}^{m} (v_j + c_w l_j w_j) u_j \tag{2.1}$$

$$\begin{aligned}
\textit{subject to:} \qquad a_j \leq u_j \leq b_j, & \qquad j = \{1, \ldots, N\} && (2.2)\\
u_j \leq \sum_{i=1}^{n} \sum_{p \in P(i)} d_{jpij} x_{pi}, & \qquad \forall j && (2.3)\\
\sum_{i=1}^{n} y_i \leq k & \qquad \forall i && (2.4)\\
\sum_{p \in P(i)} x_{pi} \leq M y_i & \qquad \forall i && (2.5)\\
y_i \in (1, 0) & \qquad \forall i &&\\
x_{pi} \geq 0 & \qquad \forall i \text{ and integer} &&\\
\forall p \in P(i), &&&\\
u_{ij} \geq 0 & \qquad \forall j \text{ and integer} &&
\end{aligned}$$

The objective function, equation 2.1, accounts for both income from piece cutting as well as a cost penalty due to waste. The first two constraints in equations 2.2 and 2.3 account for the piece type demands being satisfied. Constraint 2.4 allows for no more than k different stock rectangle shapes to be selected and constraint 2.5 restricts cutting patterns to only those stock rectangle sizes selected. The rest are integrality requirements. Beasley uses a linear program relaxation, solved via a dual simplex algorithm and guillotine restrictions to achieve results. These results were described as "reasonable," though he pointed out that it was uncertain how to quantify that term, since heuristic techniques do not locate an optimum against which the results can be measured.

Another assortment problem is addressed by Mornar and Khoshnevis [59]. They addressed the problem of the panel fabricators; the fabricators etch, laminate and mill the PCB panels for delivery to the PCCA manufacturers/assemblers. The panel fabricators have

the problem of designing multiples of panels out of large laminate sheets. Their problem formulation is similar to that of Beasley [4], with the additional problem of restricting some panels and PCB to edges of the sheets for quality purposes. They produce solutions in an attempt to minimize the number of sheets required for the demand by using a heuristic algorithm combined with a linear relaxation of the integer formulation.

Chambers and Dyson addressed the stock selection problem in the flat glass industry [8]. Multiple sizes could be selected for holding stock (only width was variable), with the objective of minimizing waste. They used a simple heuristic, in which a feasible pattern would be developed for the smallest width that produced the least waste. This process is repeated for the next largest stock width, and so on until all possible widths are explored. The widths are then traded off against each other as sets until a solution is reached which satisfies both the stock dimensional constraints as well as the piece goods demand.

The above problem may be called a "strip-packing problem." This type of problem is similar to the two-dimensional stock sizing problem in that differently-sized pieces are arranged in a stock size to be determined [14]. However, the stock is fixed in one dimension (usually designated as the width) and only the length (or height) is to be determined. This type of problem, even with the reduced dimensional complexity, is still NP complete in the general case. Coffman and Shor presented several heuristics to attack the problem.

In light of the fact that there were many stock sizing problems in the literature, Farley investigated the possibility of relationships between the stock characteristics and the cutting style for these problems [18]. Among the factors which could affect waste, the study examined: Stock place size, stock orientation, and multiple-stage cuts (number of guillotine cuts allowed). The strongest conclusion may have also be the most obvious: The larger the stock size, the less percentage of waste results. It should also be noted that the study used integer program formulations, relaxed for linear program solutions.

Response surface methodology (RSM) was applied to the "cutting stock portfolio" problem, which was essentially the two-dimensional assortment problem with a stochastic

stock length and width [22]. A Monte Carlo simulation of the demands and the stock selections were developed; response surfaces were constructed from which simple guidelines could be made for stock selection. The authors, Gemmil and Sanders, based these guidelines upon ratios of "bin length" and "average piece length." Bin length was the length of stock; for linear stock, the bin length was simply a linear dimension, but for flat (two-dimensional) stock, the bin length was actually two length measures required for rectangles. The authors found that in two or more dimensions, simple guidelines were difficult to quantify; however, the they showed that response surfaces could be constructed for specific cases to give "reasonable" results.

There are many other means by which to approach specific assortment or cutting stock problems; one author cites over 350 publications in this area [72]. The above citations are used for the purpose of illustrating the general concepts in the cutting/assortment problems.

The main formulation difficulty with cutting stock/assortment problems lies in the pattern generation. The total number of pattern combinations in a given stock size can be very large; this problem is compounded when there are a variety of stock sizes from which to choose. Of import to this panelization research is the orientation of the rectangular pieces which is not addressed at all in the assortment problem. A rectangle in a given pattern p is indistinguishable at either 0^o or rotated 180^o for a specific position. This observation is illustrated in Figure 2.2, where the highlighted locations on the PCB show how locations on the overall panel could be different for the same panel pattern.

There are several methods developed to generate the cutting patterns; most of these methods require a guillotine cut pattern, which is common in paper and glass industry [30, 12, 79]. Figure 2.3 illustrates a pattern where the cutting sequence is feasible [12]. An array of identical PCB will almost certainly be arranged in a guillotine cutting pattern as well, as shown previously in Figure 1.2. Several observations on partitioning with guillotine cuts were made by Christofides and Whitlock; these observations were made to reduce the repetitions required in cutting stock where patterns may be repeated or mirrored, relative

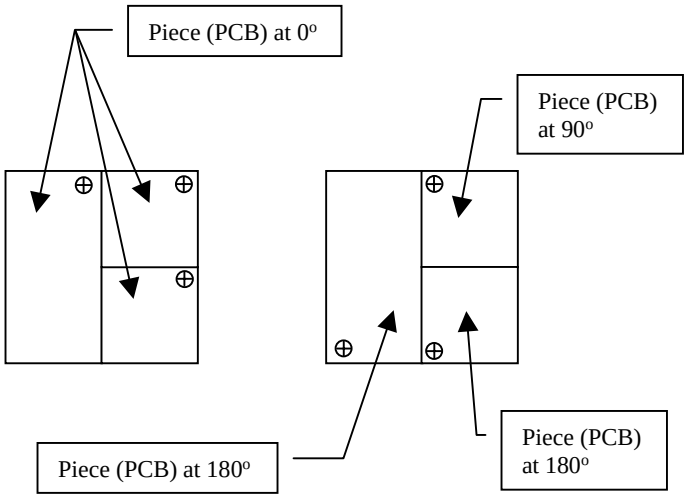


Figure 2.2: Rectangular and square pieces in an assortment problem.

to previously constructed patterns [12]. They developed a branching algorithm to partition a panel into sections of smaller, rectangular pieces in which all possible patterns would be found. The algorithm, as with many others in the literature, was used to feed cutting stock and assortment problems with the objective of minimizing trim waste.

Wang produced an additive method by which to construct partitions which ultimately grew into the panel final dimensions [79]. The algorithm was simpler to implement than the cutting/removal methods up to that time, and has been cited by many, subsequent, cutting stock researchers [4, 5, 11, 18, 28, 29, 72]. The additive method is discussed with the parameters defined in Table 2.2. There is only one set of panel dimensions, H and W, referring to the height and width of the panel. $L^{(k)}$, $F^{(k)}$, and T represent sets in which a rectangular pattern, R_i, would be collected. Waste is allowed for the patterns, both in the intermediate and final rectangle combinations, via the β_1 and β_2 percentages. A vertical (horizontal) build is considered as the placement of one rectangle shape immediately and vertically (horizontally) adjacent to a second rectangle shape. Given that the smaller rectangles (1 and 2) have widths and lengths of w_j and l_j, respectively, the new rectangle shape will have dimensions of $\max(l_1, l_2) \times (w_1, w_2)$ ($\max(w_1, w_2) \times (l_1, l_2)$) . Each fundamental piece, R_i (the PCB in the panelization problem), must not be produced more than b_i times in a given pattern.

Table 2.2: Additive pattern-building terminology[79].

k	Number of builds in a particular set in the additive pattern-building algorithm.
W	Width of panel.
H	Height of panel.
w_j	Width of a rectangle shape or a pattern build.
h_j	Height of a rectangle shape or a pattern build.
$L^{(k)}$	A set of rectangular shape build patterns, updated as part of the additive process. $L^{(0)}$ is the initial set with no builds, $L^{(1)}$ is the set after the first build is placed in the set, and so on.
$F^{(k)}$	A set of rectangular shape build patterns which is renewed as new builds are created in a new iteration of the process.
R_i	A single, temporary, pattern build composed from two other, smaller builds.
T	A temporary set which holds the R_i until they are evaluated against criteria in the additive procedure.
β_1	The allowable percentage of waste for any temporary build constructed during an iteration of build combinations.
β_2	The allowable percentage of waste for any build after all iterations are completed.

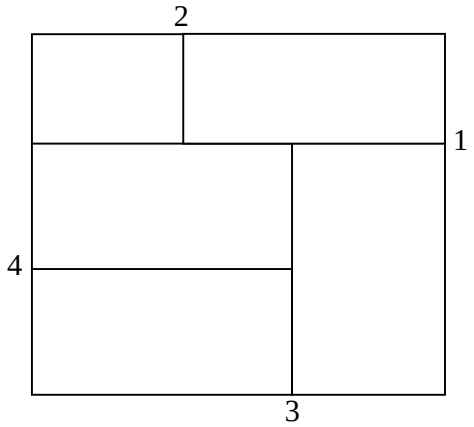


Figure 2.3: Cutting pattern made with guillotine cuts (adapted from [12]).

Step 1. a. Choose a value for $\beta_1, 0 \leq \beta_1 \leq 1$.

b. Define $L^{(0)} = F^{(0)} = \{R_1, R_2, \ldots, R_n\}$, and set $k = 1$.

Step 2. a. Compute $F^{(k)}$ which is the set of all rectangle T satisfying

(i) T is formed by a horizontal or vertical build of two rectangles from $L^{(k-1)}$,

(ii) the amount of trim waste in T does not exceed $\beta_1 HW$, and

(iii) those rectangles R_i appearing in T do not violate the bound constraints $b_1, b_2, \ldots b_n$.

b. Set $L^{(k)} = L^{(k-1)} \cup F^{(k)}$. Remove any equivalent rectangle patterns from $L^{(k)}$

Step 3. If $F^{(k)}$ is nonempty, set $k \longleftarrow k + 1$ and go to Step 2. Otherwise,

Step 4. a. Set $M = k - 1$.

b. Choose the rectangle of $L^{(M)}$ that has the smallest total trim waste when placed in the stock sheet $H \times W$.

Here k and n refer to the iterations for Step 2 and the increasing number of rectangle patterns in $L^{(k)}$, respectively. One advantage of this method is that each new rectangle shape R_i becomes a new candidate for building larger rectangles in the next k iteration. When all possible rectangle shapes are developed (when $F^{(k)}$ becomes empty), the areas of these shapes are subtracted from the panel area to find the pattern(s) with the minimum trim waste. Wang noted that no waste could be required by setting both β_1 and $\beta_2 = 0$.

As seen above, the cutting stock problem generally has the objective of minimizing the waste or the number of panels required from the cutting of stock into panels of varying sizes for a given demand. Assortment problems are a broader category of the cutting stock problem, where the objective is the same, but the number of stock sizes from which to select can be relaxed. Panelization of PCB, however, requires identical patterns for all the panels. The demand of the PCB within the panels is irrelevant, given the inherent assumption in

industry that the more PCB fitted into the panel, the less setup required for the overall demand [54, 66].

The cutting stock and assortment problems have considerable complexity involved in the formulation of the pattern set alternatives. This complexity can be considered as a result of two problems. First, an individual PCB must occupy two-dimensional space without overlapping either other PCB in the pattern or the panel boundaries. This fitting process is not linear or easy to accomplish algebraically; set theory is often mixed within linear programming techniques to formulate the constraints. If restrictions on the types of cuts are allowed (i.e., guillotine cuts), the fitting process becomes more manageable, as shown by Wang [79]. Secondly, all possible patterns are available for use in each panel produced for the demand. As mentioned earlier, such panels are allowed to be unique, or at most, a part of many sets of identical panels. In the panelization problem, all of the panels are to be produced with a single, identical pattern. Therefore, only one panel needs to be considered in the problem formulation. In section 2.1, it was noted that it is routine practice to have identical PCB in the same panel. These observations lead to the conclusion that pattern generation methods should involve practical aspects of the panelization problem when used in combination with cycle time reduce methods for PCCA manufacturing situations.

It should also be noted that a rectangular shape has two, possible orientations for a given position, and a square has four. Such positions, as suggested in Figure 2.2, are the 0^o, 90^o, 180^o, and 270^o orientations. These different positions will yield identical shapes in a fixed pattern, but produce different positions for the components within the pattern. Altered component locations within a PCB panel will yield different component placement solutions. In order to study the effect of the PCB rotations on the placement solutions, a survey of component placement research is in order.

2.3 Component Placement

As stated in Chapter 1, placement machine cycle time minimization is a popular area of research, since the majority of that time is associated with the component placement process alone. The research literature, discussed below, shows component placement time can be heavily dependent upon the sequencing of component placements and the arrangement of the component feeder devices. All of these publications make the basic assumption that the vertical pick up and placement motions of the head for each component placement are ignored. This assumption is due to the fact that each of motions require essentially the same amount of time for each component placement [3, 27, 42].

The component placement machines under consideration assemble either SMD or IPTH components. The SMD are dominating the IPTH components in industry usage, though IPTH were used in the majority of PCB manufactured until the early 1990's. IPTH components are still used today and mounted automatically with an AIM to some extent. A common IPTH component placement machine configuration will be modeled in this dissertation, because its model construction provides fundamental insight into the more complex, SMD machines and models.

Components are transferred from their bulk holding locations to their locations on the panel which is fixtured within the machine. The AIM has the components supplied via a bandolier; a schematic is shown in Figure 2.4. The SMD machines supply the components via feeder slots, which are positions located around designated, peripheral boundaries of the panel placement work zone. Matrix trays (also called waffle trays) are a flat package of shallow, rectangular depressions in an eggcrate-style which store and present the components for pick up [13]. These types of feeder trays are reserved for slower, specialty-placement machines and are not directly addressed in this research. Another type of feeder device is a feeder rack, which presents components to a robotic placement arm along a linear boundary of the component placement workspace. A third type of feeder, the feeder bank, is similar to the feeder rack, except that the bank is allowed to move linearly along its axis, so that a slot is

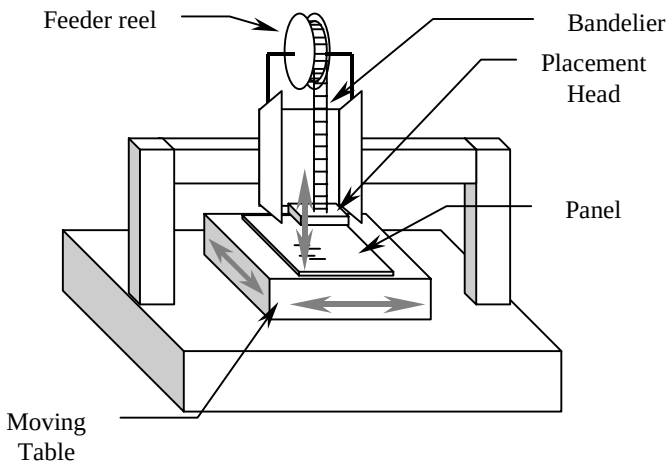


Figure 2.4: Automatic Insertion Machine (AIM) [48].

presented at the same location for every component pick-up. In the present research, "feeder racks" are considered fixed and "banks" are considered as having one degree of freedom; i.e., horizontal, linear motion. Also, some components may occupy one or more feeder slots due to their size or shape. However, in most component placement research, a one-to-one assignment is usually assumed [35, 71, 83].

Automatic insertion machines were the first, high-volume component placement machines in the PCCA industry; the use of these machines inspired the modern, industrial engineering research associated with high-volume, electronic manufacturing. A schematic of such a machine was given in Figure 2.4. Many of these machines may be described as having the components directly fed to the assembly head by a tape in a bandolier-style packaging arrangement. The component feed time may be considered as the time required to mount subsequent components in the head for the next placement; it is always assumed as shorter than the head traveling time between those sequential component insertion locations. Examples of such machines include the Amistar AI-6448, the Panasonic Panasert RT, the Universal 6287A and the Universal 6241B [48]. If the machine has multiple component types, all of these types are available in the head at the end of their respective tapes, such that

the feed time between the types is not an issue in the problem. Therefore, all components placed in the AIM machines may be viewed in terms of the same mathematical formulation.

The problem of minimizing the placement time in an AIM thus becomes one of minimizing the traveling time of the head as it travels between the component locations. The vertical descent and rise of the placement head is identical for each of the component placements; such time is also considered as a small, relative to the planar component location travel time. Thus, it is ignored in minimization problems [3]. The head planar velocity is considered as a constant value in the minimization problem; this assumption is common in the literature [3, 42].

It should be noted that the AIM usually has a fixed head position and the board beneath it moves to each subsequent x-y location instead. However, the problem formulation will be the same, and discussion of the head movement, relative to the panel, is convenient for subsequent discussions of other more complex machines.

With the above assumptions, the AIM component placement problem may be considered as the classic Traveling Salesman problem (TSP) [45]. The TSP considers distances as the primary cost function to be minimized. If the head planar velocity is considered as constant, then distance between locations may be considered as the objective function in the AIM problem as well.

A generic panel selected for placement is shown in Figure 2.5. In this illustration, panelization of the panel (i.e., identification of multiple PCB within the panel) is not required. Let the component i location be specified in rectilinear coordinates as (x_i, y_i) from an arbitrary origin of the panel's lower left-hand corner. A component placement sequence may be considered as the placement of component j immediately following the placement of component i. The distance, d_{ij}, is then given as

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2.6}$$

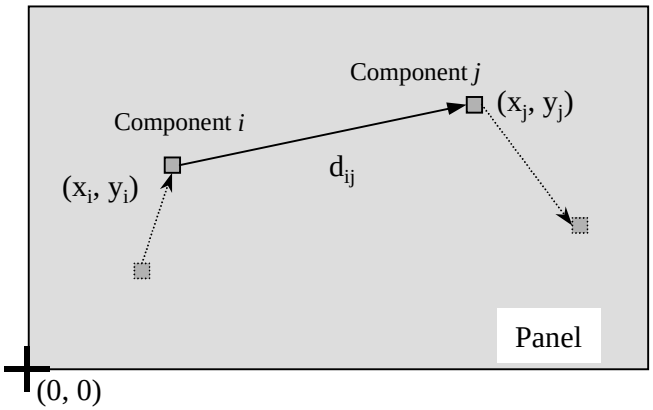


Figure 2.5: Generic panel (no panelization distinction).

This distance metric is often described as the Euclidean metric; it will be the metric used throughout this research [3, 42, 83].

With a distance metric defined, the TSP problem formulation can be stated as the following:

$$\textit{Minimize:} \quad \sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}u_{ij} \tag{2.7}$$

$$\textit{subject to:} \quad \sum_{j=1}^{N} u_{ij} = 1; \qquad \forall i \leq N \tag{2.8}$$

$$\sum_{i=1}^{N} u_{ij} = 1; \qquad \forall j \leq N \tag{2.9}$$

$$z_i - z_j + Nu_{ij} \leq N - 1; \qquad \forall i, j = \{2, \ldots, N\} \tag{2.10}$$

$$u_{ij} \text{ binary}$$

In equation 2.7, u_{ij} is a binary decision variable which is 1 if component i is placed before j and 0 otherwise. The constraint of equation 2.10 is required to eliminate any subtours. The formulation is the classic TSP linear program, which is considered as NP-hard, for which

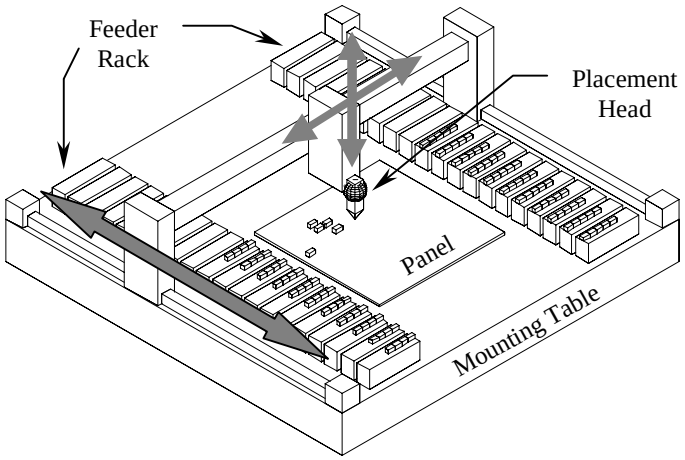


Figure 2.6: Pick-and-place machine (PAPM).

large problems (exceeding 273 locations), optimal solutions have been difficult or impossible to find [45]. Since components on panels can number in the hundreds, heuristics have been used to find good placement time solutions in the research literature. Such an approach was used by Chan and Mercier in the solution of their component insertion machine problems [9].

Most papers usually cite Ball and Magazine as the first, organized research on component placement time reduction which included both component sequencing and feeder arrangements [3]. The PCB component sequencing problem was modeled for a moving-head, stationary-board, stationary-feeder machine. This type of machine frequently is identified as a pick-and-place machine and is illustrated in Figure 2.6 [78, 42, 48].

An interesting observation by Ball and Magazine was that, for a pick-and-place machine, the possible paths required to populate the board could be segregated into those which were *required* and those *nonrequired*. As shown in Figure 2.7, the required arcs were those from feeder slots assigned to component types needed on the board. The order of component placement sequencing would determine the nonrequired arcs. With the feeder slot assignments considered predetermined, the problem was to find the optimal component sequencing

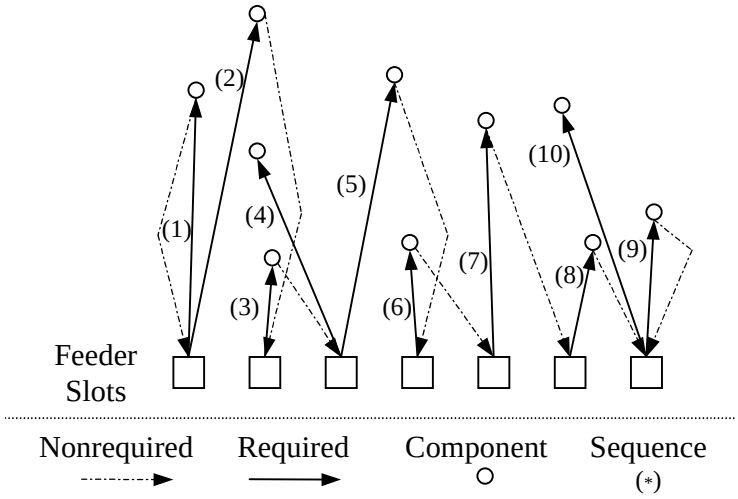


Figure 2.7: Required and nonrequired arcs for a PAPM [3].

of the components to minimize the head travel time. The machine movement under these conditions was modeled as a rural postman problem, which is NP-hard. The rural postman problem is to find a closed path such that each member of required arcs appears exactly once in the path and the cost of the closed path is minimized. Ball and Magazine addressed the problem with a heuristic which first used a relaxed form of the mathematical network formulation, then uses a minimum spanning tree technique to connect the network produced from the first step.

Grotzinger and Sciomachen used Petri nets to characterize the turret-head, high-speed placement machine early in the development of this genre of research [26]. The characterization in [26] was a mathematical description of the machine's operation, for which Grotzinger and Sciomachen did not attempt to optimize in terms of placement time. The characterization was also one of the few publications in which the vertical pick up and placement motions were included as part of the formulation.

Another early research publication addressing the combined component sequencing and allocation problem was presented by Leipälä and Nevainen [46]. They investigated the movements possible from the Panasert RH machine; schematically represented in Figure 2.8,

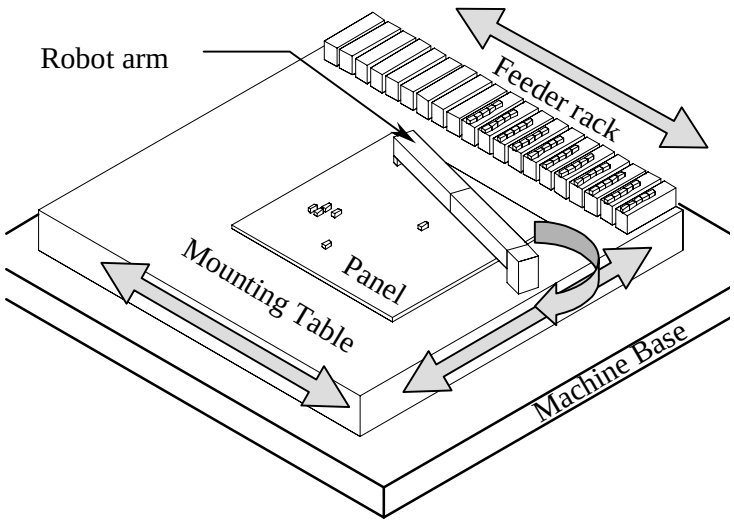


Figure 2.8: One head, rotational robot arm inserter.

the machine movements consist of linear feeder movement, x-y planar panel movement, and rotational motion from the robot placement arm. The authors recognized the problem as NP-hard and divided it into two parts: Component placement and feeder allocation. The component placement problem was considered as a traveling salesman problem, given an initial feeder allocation set. The feeder allocation problem was then addressed as a quadratic assignment problem for a final solution.

Ahmadi et. al. analyzed the component allocation part of the placement problem in a dual-head placement machine [1]. The machine under investigation (schematic in Figure 2.9) used two independently-moving heads, picking from opposing feeder racks, to place components on a panel which moves underneath the placement heads. The problem is additionally complicated in that, due to the dual heads, the controller must anticipate the wait time of one head for placement while the other is placing a component. Recognizing the complexity of the problem, the authors focused upon separate issues involving component allocation. They first addressed the higher-level issue of assigning *multiple*, identical, component-type reels to the machine in order to complete as many panels as possible before reloading the feeder with reels again. Once that problem was solved (with a mixed integer programming

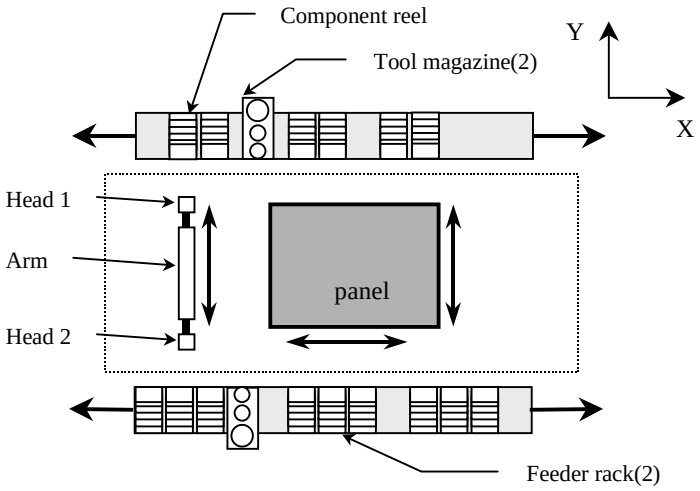


Figure 2.9: Dual head placement machine (adapted from [1].

formulation), they then addressed the assignment of such component reels to the feeler racks' slots. Throughout this process, they assumed that the placement sequence was fixed, and therefore not included in the solution process.

In a separate publication, Ahmadi and Kouvelis continued with research on the dual-head placement machine, expanding the allocation problem to include an approach for the single and multiple product staging problem [2]. A branch and bound procedure was used with a Lagrangian relaxation of an IP formulation. For the multiple product staging problem, a heuristic procedure, utilizing a greedy knapsack algorithm, was developed to produce "optimal" solutions, though the solution method clearly based upon heuristic techniques.

Ji et. al. used an assignment-based approach to handle a PAPM problem [40]. They accounted for possible nozzle (tool) changes, waiting time for a feeder to present the next component in a reel, and variable placement head speed. The illustration of the assignment problem was accomplished with bipartite graphs and cost matrices were constructed to estimate time costs for assignments to the sequencing of component placement, given a feeder arrangement. Linear relaxation was used to determine a placement sequence for the problem.

Recognizing the complexity of addressing multiple panel designs produced on the same machine, Haung, Srihari, Adriance and Westby used knowledge-based heuristics to produce component placement sequences and feeder assignments for multiple, dissimilar panels [36, 35, 69]. Their work also incorporated multiple batch release solutions with the purpose of reducing the overall production makespan for a given, periodic demand requirement. In particular, the Universal Instruments Model 4766A machine was used for the research. This machine is a PAPM with two, separate workspaces; each workspace has its own placement head and fixed feeder racks. The panel is moved to the first workspace, where high-resolution component placement is required with the high-accuracy head. Once work is completed in that workspace, the panel is automatically moved to the second workspace, where a "more flexible" head places components of various types. The knowledge-based rules included some general concepts of:

- *Batch Sequencing.* The larger the batch size, the less important the batch sequencing.
- *Feeder Arrangement.* Basically, the more numerous the component type, the more importance will be assigned to placing that type in a feeder slot close to the components' locations.
- *Tooling Arrangement.* This concept accounted for additional time required for tooling/nozzle changes.
- *Traveling Path Improvement.* The TSP considerations discussed earlier are considered here in a knowledge-based format.
- *Panels and* 180^o *Offset Boards.* In this context: Smaller, identical PCB can be arrayed as a larger panel to reduce setup times. "Offset boards" refer to the practice of arranging two PCB on a single panel in 180^o orientations, relative to each other. This practice is solely for the purpose of improving yields from reflow process for some troublesome component arrangements due to better heat distribution.

Table 2.3: Four rules used in the knowledge-based approach by Yeo and Yong[84].

Rule 1.	Sequence component placement in a path that takes the minimum time.
Rule 2.	Arrangement of components in the feeder that provides minimum pick-up time.
Rule 3.	Mount identical components in one pass.
Rule 4.	Sequence component placement according to component size.

The topic of bullet 5 is one of the very few in the literature where panelization concepts are addressed in component placement issues. The authors did not elaborate further on any of these loose categories of rules; instead, a discussion of the computer program structure was the main emphasis of their research. Shih, Srihari and Adriance later conducted further research into the actual implementation of the heuristics via an artificial intelligence-oriented computer language called PROLOG [69].

Another knowledge-based approach to the placement sequencing and feeder arrangement problem was presented by Yeo and Yong [84]. They described their work as a frame-based approach in which rules were applied to reduce the cycle time on a rotary turret machine with moving feeder racks, or a "Fuji Machine." This type of machine is illustrated in Figure 2.10. The four rules, shown in Table 2.3, were used to guide an object-oriented software program to reduce cycle times. Additionally, a part of heuristics, common in industry, were a part of the program. The first, the Constant Incremental Feeder Heuristic, is used in the attempt to keep feeder rack pick-up movements down to one adjacent slot at a time, if possible. The second, the *Nearest Neighbor Heuristic*, attempts to sequence each subsequent component placement to the location nearest to the last placement location (within a user-defined tolerance).

The rotary turret machine was addressed by a knowledge-based approach, but combined with TSP algorithms to provide input into the component placement sequencing

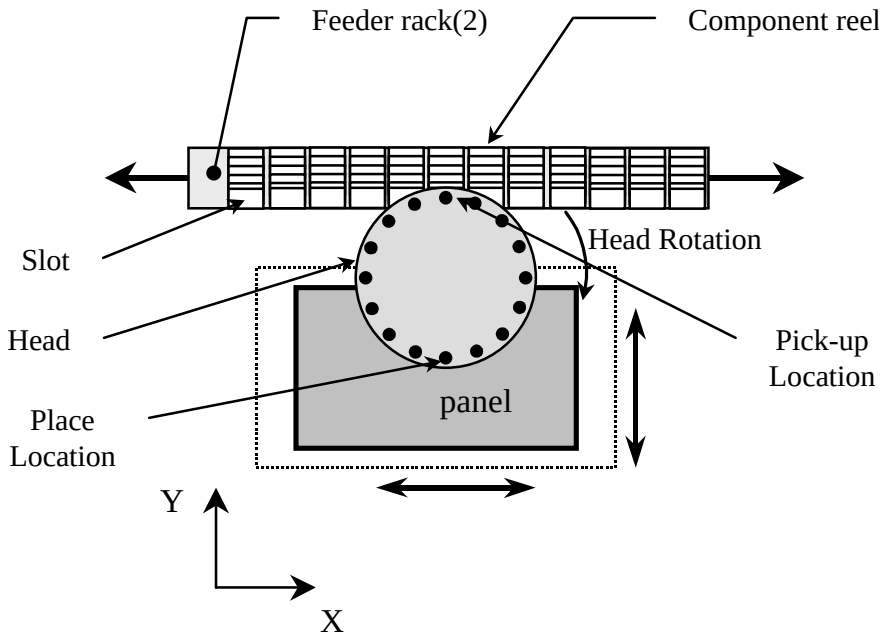


Figure 2.10: Rotary Turret Head (RTHM) machine schematic.

[16, 15]. This machine has a fixed-location turret with 16 heads that rotate between a linearly moving feeder rack and an $x-y$ board placement table, essentially the same as that in Figure 2.10. The authors noted several practices of the proprietary placement algorithm, among which were the efforts to prioritize feeder assignments to those components of the smallest size and greatest quantity first. The turret head moved fastest with the smallest sized components; its speed must be reduced for larger components to reduce the chance of a component flying off the head before placement. Thus, an important observation was that, with simultaneous pick-up and placement occurring, the largest time between pick-up/placement events was usually due to motions of the X-Y table. The authors concluded that they could reduce the turret head waiting time (for the moving table) by constructing an intelligent, knowledge based program for that particular machine. Their program would attempt to more evenly distribute the $x-y$ motion distances between placements through the interaction of a TSP-based subset of the software.

The PAPM was modeled by Broad et. al.; both the component sequencing and the

component/feeder allocation problems were addressed [6]. The authors used a graph theory approach to a model a Dynapert MPS 500. In this machine, a single robot head placed components from feeder locations at either side of the PCB; a tool change is required for component types occupying different feeder widths. Two solution methods were used; the first method used the integer-programming (IP) model for the placement sequencing problem and solved this Traveling Salesman Problem (TSP) [45]. Once the TSP was solved, the components were allocated to feeders by an algorithm based upon the sequence. The second solution method was to iterate between successive TSP solutions and a pairwise exchange of feeder allocations for continually improved, overall placement times. These sub-optimal solution techniques were used to gain quick, useful solutions to the NP-hard mathematical formulations. In this case, the IP formulation included only partial TSP subtour constraints, such that a patching algorithm was necessary to repair sequence solutions which generated such subtours.

In [6], a case was made that the component assignment problem can be separated from the entire problem without loss of optimality in the solution process. This assertion was likely overstated on the authors' part, based upon the fact that other authors in the field subdivided similar machine problems with the caveat that optimality *was not* assured under such a procedure [3, 42, 48]. Furthermore, the TSP IP was not solved optimally since subtours were only partially eliminated and patching routines used in such occurrences. The authors did not prove optimality of their approach; hence, it is likely that their "optimal" solutions were only local optima. Nevertheless, improvements of 10% to 25% in the placement times were reported over that of the subject company's component placement sequencing and allocation manual methods.

Kumar and Li analyzed a Quad Systems PAPM [42]. The authors understood that the optimal solution to a problem requiring both the placement sequencing and component allocation was NP-hard. Thus, they divided the problem into two decoupled problems and solved them separately. The two subproblems consisted of the TSP for the component sequencing and a minimum weight matching problem (MWMP) for the feeder assignment.

However, their integer programming formulation was partitioned such that the TSP formulation addressed those "required" motions discussed by Ball and Magazine (Figure 2.7) [3]. The MWMP solved addressed the remaining paths, the "nonrequired" motions. Their approach (using Lagrangian relaxation) yielded solutions that were optimal for each subproblem, but not optimal for the larger problem. Interestingly, the solution produced from the subproblems was not used as the final result; instead, the solution was the seed for further reduction techniques (2-optimality and 3-optimality methods) for NP-hard IP formulations [45].

Kumar and Li used three heuristics, common in the PCCA industry, against which they compared their solutions. These heuristics are cited by other researchers as a standard and will therefore be addressed here: The typewriter, the S-shape and the largest candidate rule [46, 16, 48, 49, 36]. The first two methods assign component locations in a placement sequence; the third method assigns component types to appropriate slots in the feeder bank. In both the typewriter and the S-shape methods, the panel components with a smaller y-coordinate value are placed before those of a higher value. The type-writer method further requires that, for those components of the same y coordinate value, the placement sequence is in increasing x-coordinate values. For the S-shape heuristic, for those components of the same y coordinate value, the placement sequence is in order of increasing x-coordinate values, then decreasing values for the next row, and alternate thereafter. An example is of each is illustrated in Figures 2.11 and 2.12.

The largest candidate rule is used to assign component types to feeder slots. The component type with the greatest number of components on the panel is assigned to the slot nearest the middle of the panel's edge. The component type with the next-greatest number of locations on the panel is assigned adjacent to the first, and so on. The theory behind this method is that the slot which is to be accessed the most frequently by the placement head should be located such that it can be accessed most quickly from any potential location on the panel. A pair-wise exchange process is the swapping of one component type assignment to a slot with another, then recalculating the objective function. The exchange process is

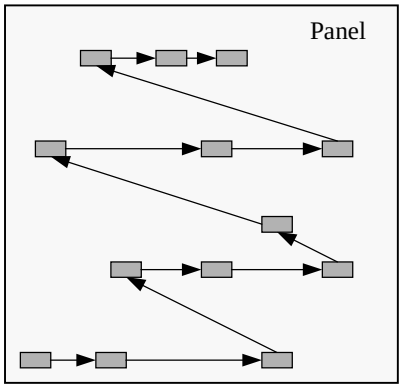


Figure 2.11: Typewriter heuristic example.

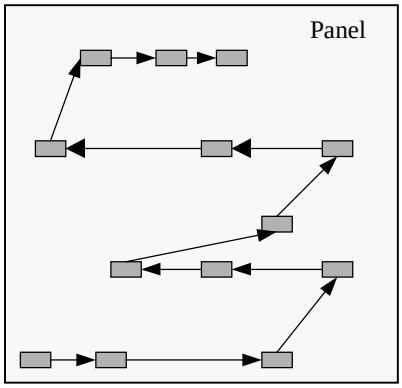


Figure 2.12: S-shape heuristic example.

continued until no further improvement is obtained from an exchange. These component assignment processes are not used for AIM machines, of course.

For specific machine types, algorithms can be developed which produce better placement time solutions than those algorithms which are created to handle a broader array of machine types. This assertion was proved by Moyer and Gupta [61, 60]. They developed the acyclic assembly time (AAT) algorithm specifically for the RTHM, keying in on the fact that the RTHM delays between component placement was due to waiting of one of the three primary moving subsystems: The rotary head, the feeder rack or the panel (mounted upon a moving table). Moyer and Gupta pointed out that the rotary head need not increment one index position for every pick-up, nor for every placement, as had been assumed in the research literature thus far. Better results could be obtained if the head were allowed to pick-up at every opportunity on the feeder rack, as either the feeder rack paused or the board paused for a placement motion. Similarly, the head would be allowed to place a component at similar opportunities. They created a complex algorithm to incorporate such asynchronous motions which was compared against the published research literature for RTHM problems. Their algorithm contained subroutines which solved portions of the TSP (for component placement sequencing) and quadratic Assignment Problem (QAP) (for feeder rack slot allocation) as part of a complex computer program which continually improved interim solutions. The AAT algorithm proved to reduce the assembly time for those examples by at least 24%. The major drawback to the AAT algorithm is that it is valid *only* for the RTHM.

The above literature addresses the problem of reducing placement times in existing machines. Another alternative to this problem is to propose new assembly machine types. Chang and Young created a concept whereby simultaneous placement of multiple components are allowed [10]. This concept was a fundamental shift in the paradigm of placing components, since even dual-head and rotary turret head machines still place components on the panel in a sequential fashion. The proposed machine would utilize a “gang” assembly mechanism, holding many heads which pick up components simultaneously.

A different machine concept was suggested by Su et. al. [71]. A placement head would have two degrees of freedom (DOF) over the working plane, where one DOF would be along a linear path across the panel (traditionally considered the y axis) and the second DOF would be the vertical motions required to pick and place each component. In addition, the feeder rack would have linear motion along what has been defined as the x axis. And lastly, the worktable, upon which the panel was affixed, would also have x-y motion capabilities. The first alternative, shown in Figure 2.13, restricted the robot to pickup at only one designated place in the feeder rack linear movement range. The placement onto the board was also restricted to occur at only one place relative to the machine base. This machine alternative was designated as the fixed PAPM (FPP) model and was similar to many PAPM in industry. The second and new alternative shown in Figure 2.14, the dynamic PAPM (DPP) model, allowed for a pickup anywhere along a linear path where the feeder could move. The placement process would also be dynamically determined, in the sense that both the worktable and the head would meet anywhere in the worktable movement region. The FPP model was more restrictive but easier to address in problem formulation that the DPP model. The DPP model would have the additional complexity of determining the pick up and placement locations for each component on the panel, but may yield better cycle times than the FPP. The authors used design of experiments techniques to study the advantages of one model over the other. The conclusion was that the DPP model outperformed the FPP model for nearly all cases, though the DPP concept would require advances in the technology of dynamics and control applied to SMT placement machines.

The automatic placement of odd-form components has been investigated to some extent; odd-form components are those which do not fit into the shape characteristics which allow rapid placement with either standard SMD or IPTH machines. These types of components are assembled either by hand or with more general-function SCARA robots. Lin et al. investigated the problems of using two such robots to place components on a single panel, shared between the two [50]. They examined two concepts:

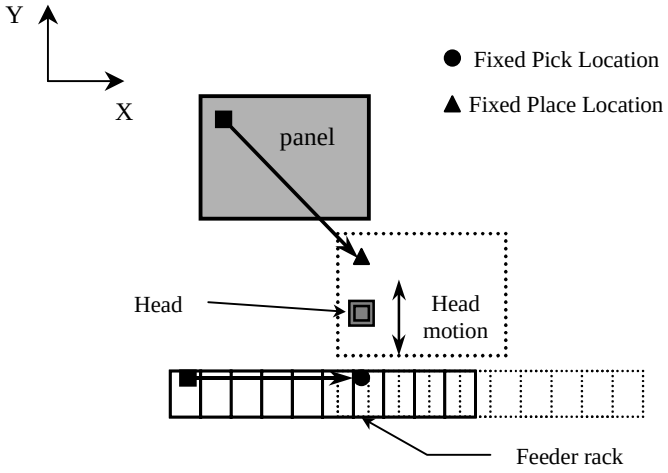


Figure 2.13: FPP model (adapted from [71]).

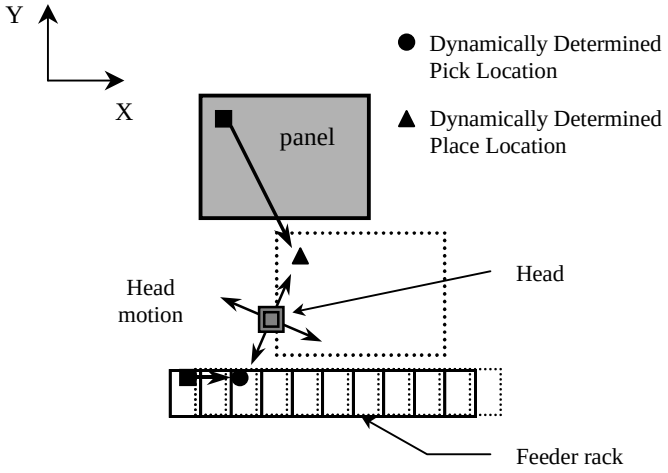


Figure 2.14: DPP model (adapted from [71]).

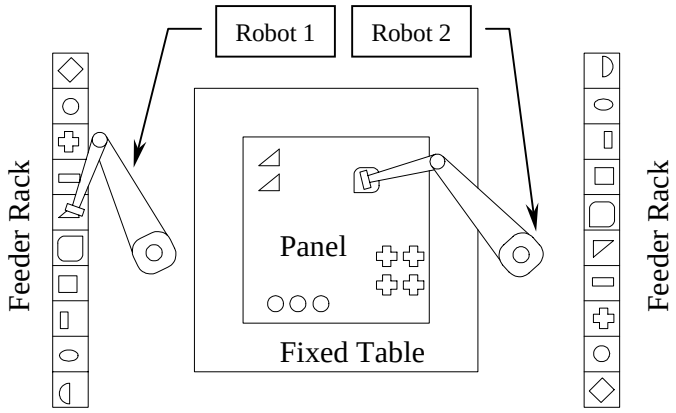


Figure 2.15: Layout of two-concurrent-robot assembly workstation (adapted from [50]).

1. Figure 2.15: Two opposing robots sharing a common workspace, upon which a board was mounted, and

2. Figure 2.16: Two opposing robots utilizing a shifting pallet which would position itself under one robot for placement, while the second robot picks up the next component.

The main purpose of their research was to show how a two-robot placement workstation was superior to a single-robot placement workstation. It also illustrated how researchers could develop ideas outside the current marketplace machine-type paradigms to improve placement times.

2.3.1 Common Traits in the Machine Types

The motions associated with the placement machines can be categorized as either placement (robotic) arm movement, horizontal panel motion and feeder bank linear motion; the arm movement could be either Cartesian or rotational or a combination. Not all of these motions are present in all types of machines.

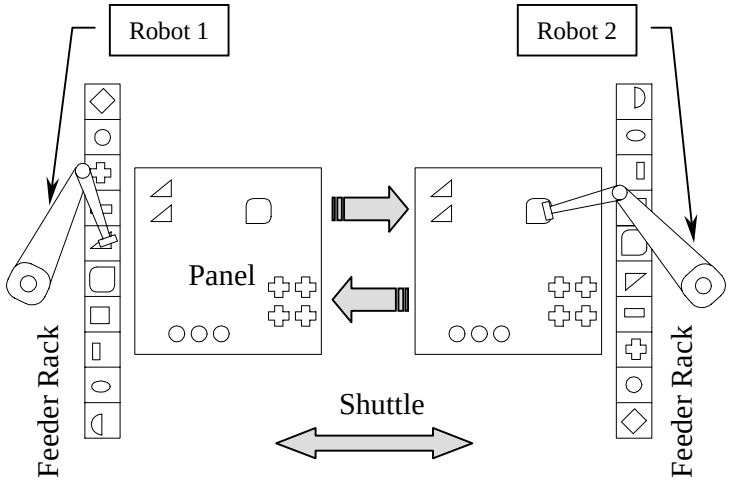


Figure 2.16: Layout of two-robot sequential assembly workstation (adapted from [50]).

The research literature discussed above assumes that the component types on a panel have already been assigned to the particular machine in question; i.e., the higher-level, production planning for this particular panel, across any placement machines in the production line, has been solved in an analysis separate from and prior to this one. For PCCA production planning research, the reader is referred to [47, 52, 53].

The mathematical formulation of the component placement problem is dependent upon the type of machine being modeled. Though there are a great deal of machine manufacturers with different types of machine placement configurations, this research identified three basic types of machine configurations. The rationale for such a limit is twofold:

- Past research and publications frequently use these three types of machines as basic placement configurations.
- The purpose of the research is to establish panelization as a design parameter which can improve the placement times of products. The implementation of the concept, exhaustively applied to all placement machines currently in existence, is not necessary for the introduction of the new design parameter concept.

The component sequencing and allocation problem has thus far been addressed as a combinatorial, NP-hard problem, whereby researchers have found solutions through approximate algorithms. The combinatorial type of problem fits well into solution methods which use *genetic algorithms* (GA) and thus the component placement problem has been addressed with GA techniques [48]. Li, Wong and Ji have produced a series of publications which address the placement time optimization of several types of placement machines using GA [48, 83, 49]. An overview of GA will introduce this technique so that the reasons for its application to combinatorial problems can be discussed.

2.4 Genetic Algorithms

Over the past two decades, the genetic algorithm technique has become a popular means by which to produce satisfactory solutions for computationally intractable problems [55, 20, 56]. The GA was originally presented by Holland in 1975 [32]. This technique is a stochastic optimization method which is conveniently presented via the metaphor of natural evolution. This metaphor is presented in the context of randomly initialized individuals which form a population; this population evolves in the fashion of the "survival of the fittest" over successive generations. "Children" are created using simulated genetic operations such as mutation, crossover or other methods. The survival of the newly generated children (solutions) depends upon how well they are measured in comparison to their siblings; this measure is called the *fitness* of the individual. A portion of the individuals from each generation with the best fitness values are kept with a high probability; the worst are discarded.

Genetic algorithms search the solution space of a problem through the use of simulated evolution, i.e., the survival of the fittest strategy. In general, the fittest individuals of any population tend to reproduce and survive to the next generation, thus improving successive generations. Also, inferior individuals can, by chance, survive and also reproduce. Genetic algorithms have been shown to produce solutions for linear and nonlinear

problems by exploring all regions of the state space and exponentially exploiting promising areas through mutation, crossover, and selection operations applied to individuals in the population [55]. A genetic algorithm (GA) is summarized below:

1. Supply a population S_0 of Q individuals and respective function values.
2. Operate the *selection function* upon the individuals.
3. Operate the *reproduction functions* upon the selected individuals in the mating pool to produce children.
4. Merge children into the original population.
5. Operate the *evaluation function* upon the new population.
6. Repeat steps 2–5 until *termination criteria* is reached.
7. Report results.

Six issues should be addressed when using GA: solution representation, selection function, genetic operators (i.e., reproduction functions), population initialization, evaluation function, and the termination criteria [55, 34].

2.4.1 Solution Representation

Each individual in the population must be represented by a chromosome representation. The representation scheme determines how the problem is structured in the GA and also influences which genetic operators may be used. Each individual or chromosome is made up of a sequence of genes. The gene representation could consist of binary digits (0 and 1), real numbers, integers, matrices, etc. Originally, GA genes were limited to binary digits, but it has been shown since then that natural representations of various data formats are more efficient and produce better solutions for some problems [32, 55]. Therefore, the use of integers to represent genes in TSP, scheduling and sequencing problems has become popular.

Of interest to this research is the path representation of tours in a TSP problem [55]. For example, a tour of

$$1 \Rightarrow \quad 3 \Rightarrow \quad 5 \Rightarrow \quad 2 \Rightarrow \quad 4$$

is represented in a chromosome as

1	3	5	2	4

The representation of a chromosome guides, to some extent, how GA operations are performed. Additionally, problem-specific operators are usually required [7, 75, 37]. These operators will be discussed in Section 2.4.3.

2.4.2 Selection Operation

The selection of individuals to produce successive generations is based on some sort of probabilistic method and is based upon the individual's fitness. All individuals have a chance of being selected; an individual may even be selected more than once. Various methods exist to assign probabilities to individuals; the most popular are the roulette wheel and tournament ranking. The roulette wheel, developed by Holland [32], was the first selection method. The probability, p_i, for each individual is defined by:

$$p_i = \frac{F_i}{\sum_{j=1}^{\text{PopSize}} F_j}$$

where F_i equals the fitness of individual i. The population is not required to be ranked or ordered before the cumulative probabilities are calculated.

Tournament selection requires the evaluation function to map solutions to a ordered set, however, it does not assign probabilities. Tournament selection works by selecting a set

number of individuals randomly, with replacement, from the population, and then inserting the best of these into the mating pool. This procedure is repeated until a predetermined number of individuals for the mating pool have been selected. In this method, the differences in magnitude of the individual chromosomes are not a factor in the selection process; only their relative order in the list is of importance.

An elitist selection technique can be integrated to both schemes. In such an implementation, the fittest individual (or a range of individuals) of each generation is allowed to be placed in the mating pool, regardless of whether or not it was selected by the selection process. A last matter, related to selection, is that of the "generation gap." The generation gap is defined as the proportion of individuals in the population which are replaced in each generation. Most work has used a generation gap of 1; i.e., the whole population is replaced in each generation. This value is supported by the investigations of Grefenstette [25]. However, a more recent trend has favored steady-state replacement [80, 81, 74, 73]. This practice allows a proportion of individuals in each generation to be replaced. In the steady-state case, the mating pairs must be determined as well as the pairs from the original population which will be replaced (originally, *all* were replaced by the children).

Several schemes are possible, including:

1. Selection of parents according to fitness, and selection of replacements at random.
2. Selection of parents at random, and selection of replacements by least fitness.
3. Selection of both parents and replacements according to fitness or least fitness.

Whitley's GENITOR algorithm, selects parents according to their ranked fitness score, and the children replace the least fit population individuals [81]. This technique apparently improves the GA performance for TSP; however, it should be noted that Goldberg and Deb found that the advantages claimed for steady-state selection could be obtained by using exponential fitness ranking, or large-size tournament selection [24].

2.4.3 Genetic Operators

Genetic operators provide the basic search mechanism of the GA. The operators are used to create new solutions based on existing solutions in the population. Research has produced many operators, but they all may be considered as originating from two basic types: mutation and crossover [55].

Mutation randomly selects one gene out of a selected individual and changes the value to another feasible value. For a gene in a binary chromosome representation, the change is simple: A mutation results in $0 \rightarrow 1$ and $1 \rightarrow 0$. For integer or real-valued gene representation, the mutation is usually a random selection of the existing value to a new value within the feasible range.

Crossover takes two individuals and produces two new children; mutation alters one individual to produce a single new chromosome configuration. The application of these two basic types of operators and their derivatives depends on the chromosome representation used.

A diagram of a simple crossover (between binary chromosomes) is illustrated below. A random cut point is selected between two genes in the selected, parent chromosomes ($\tilde{s}_{P1}$ and $\tilde{s}_{P2}$). The two pieces are then swapped between the parents. The resulting two, new chromosomes are the offspring ($\tilde{s}_{C2}$ and $\tilde{s}_{C2}$) of the operation.

$$\tilde{s}_{P1} = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$$

$$\tilde{s}_{P2} = \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}$$

$$\tilde{s}_{C1} = \boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}$$

$$\tilde{s}_{C2} = \boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}$$

Problems can arise with certain problem structures and chromosome representations. For example, if one were solving a TSP problem with a path representation, simple crossover could be represented as below. However, the parents would produce offspring containing subtours as a result of repeated numbers. More importantly, some locations could be omitted from the chromosome solution representation; these solutions would therefore produce an infeasible TSP path representation.

$$\tilde{s}_{P1} = \boxed{1}\boxed{3}\boxed{5}\boxed{2}\boxed{4}$$

$$\tilde{s}_{P2} = \boxed{5}\boxed{4}\boxed{1}\boxed{2}\boxed{3}$$

$$\tilde{s}_{C1} = \boxed{1}\boxed{3}\boxed{1}\boxed{2}\boxed{3}$$

$$\tilde{s}_{C2} = \boxed{5}\boxed{4}\boxed{5}\boxed{2}\boxed{4}$$

Specific crossover operators have been developed to handle such a problem. One such example, the CX operator, builds offspring without a randomly selected crossover point, but does result in feasible TSP solutions [55]. Using the parent chromosomes above, the first location would be taken from the first parent.

$$\tilde{s}_{C1} = \boxed{1}\boxed{\text{x}}\boxed{\text{x}}\boxed{\text{x}}\boxed{\text{x}}$$

The next gene selected would correspond to the value in the $\tilde{s}_{P2}$ chromosome which corresponds to that value in the $\tilde{s}_{P1}$ location where the first gene swap occurred. This value, 5, would be placed in $\tilde{s}_{C1}$ at the same location as where it is in $\tilde{s}_{P1}$.

$$\tilde{s}_{C1} = \boxed{1}\boxed{\text{x}}\boxed{5}\boxed{\text{x}}\boxed{\text{x}}$$

The process is continued until a value, selected for placement in $\tilde{s}_{C1}$, would be repeated in that chromosome. In this case, this situation occurs after 5 is selected, because *1* would be

the next value to be placed. At this point, the remaining spaces in $\tilde{s}_{C1}$ are filled in with the values from $\tilde{s}_{P2}$ for the corresponding locations. The same process is repeated for $\tilde{s}_{C2}$ to produce two, new offspring.

$$\tilde{s}_{C1} = \boxed{1}\boxed{4}\boxed{5}\boxed{2}\boxed{3}$$

$$\tilde{s}_{C1} = \boxed{5}\boxed{3}\boxed{1}\boxed{2}\boxed{4}$$

The GA operators have been discussed thus far in the context of being applied at a constant rate over each generation. However, recent research has shown that some problems yield better solutions if the rates of the operators are varied, or *adapted*, to the success of particular operators over successive generations [25, 19]. This approach assumes that multiple types of operators are applied to the population on every generation.

2.4.4 Initialization, Termination, and Evaluation

The GA must be provided an initial population. One method is to randomly generate solutions for the entire, initial population. However, since GAs can iteratively improve existing solutions, the initial population can be seeded with potentially good solutions from any heuristics available. The remainder of the population may then be randomly generated.

The GA iterates to successive generations until a termination criterion is met. The most frequently used stopping criterion is a predetermined, maximum number of generations, while other strategies involve convergence criteria associated with population fitness values [55]. When the sum of the deviations among individuals in a generation becomes smaller than some specified threshold, the algorithm can be terminated. Alternatively, the lack of improvement for the best fitness between a successive number of generations can result in termination. Lastly, a target value for the evaluation measure can be established based on some arbitrarily acceptable threshold. Several of these strategies can be used in conjunction with each other to produce termination under different circumstances.

The evaluation function is used in a GA and is subject to the minimal requirement that the function can map the population individuals into a representative set. The evaluation function is independent of how the GA (i.e., stochastic decision rules) are applied, but is a result of the problem formulation to be solved. An evaluation function will use the chromosome information to produce a level of performance for that solution. In the case where multiple chromosomes represent the solution, the objective function must transform the chromosome set information into a single, quantitative value.

Many NP-hard optimization problems have been addressed using GA in recent years [33, 64]. There are several reasons GA is popular for these situations. First, they do not have stringent, mathematical requirements of their use relative to the optimization situation. GA techniques have been applied to linear problems, nonlinear problems, discrete, continuous or mixed valued situations. Another advantage the GA method lies in the fact that it does not become fixed within a local optimum region of the solution space, due to its stochastic nature.

2.4.5 GA Use in Cutting Stock and Component Placement Research

Gemmil used GA to solve the one-dimensional assortment problem [21]. He addressed the problem of determining appropriate stock lengths for a demand of pipes. The GA population individuals were coded into binary data chromosomes, as was customary of the more traditional GA encoding practice. Crossover and mutation were the operators used and a roulette wheel selection scheme was enacted. Gemmil compared his results for specific problems to solutions produced by Beasely's heuristic [4]. He concluded that the GA produced better solutions than the heuristic as the number of possible stock sizes were increased.

The application of GA to a two-dimensional assortment problem was accomplished by Ismail [38]. These shapes were odd-shaped polygons with the requirement that their corners

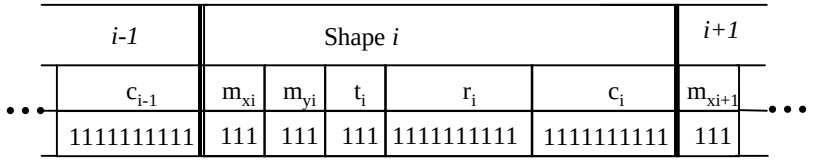


Figure 2.17: Representation of pattern position and orientation [38].

must be right angles. For example, a rectangle or a cross would fit into this category. The shape was defined by its occupation of a two-dimensional grid; if the shape fell across a grid square, that location on the grid had a value of *1*. Otherwise, the grid square had a value of *0*. After fitting as many shapes as possible onto the grid, the grid square values were summed, with the objective of having as large a summation value as possible. The genetic encoding scheme required identification of each of the shapes in a single, long chromosome, as shown in Figure 2.17. The values r_i and c_i represented the binary horizontal and vertical position of the left-hand corner of shape i in the grid, in grid units, respectively. The m_{xi} and m_{yi} indicated the mirrored shape of the object along the x-axis, the y-axis, respectively. The rotation, either 0^0 or not 0^0 (i.e., 90^0), was indicated by t_i. The three-bit representation of the last three parameters were "found to increase the chances of selecting and modifying the orientation parameters." Each three-bit parameter was summed; if less than three, then the parameter were true and false otherwise. Specialized GA operators were developed for this problem and the procedure produced results for test cases that were up to 85% better than random search methods [38]. However, the author noted that the encoding scheme resulting in very long computation time requirements.

The combinatorial problems associated with the component placement time problem is a natural application for GA. Several researchers have published research in this area in the past decade [48, 49, 83, 17, 7]. Their work gives insight into the construction of chromosomes

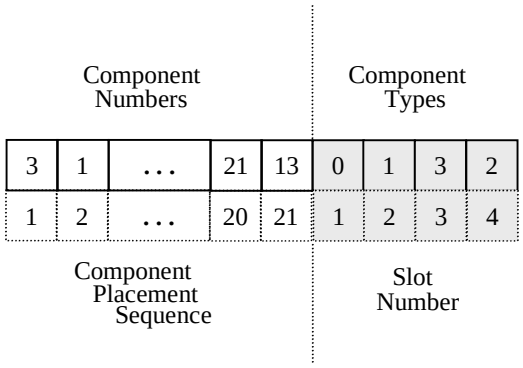


Figure 2.18: Example of a 21 component, 3 component type chromosome [48, 49, 83].

for this type of problem. In particular, the AIM, PAPM and RTHM were addressed with GA by Leu, Wong, and Ji [48, 83]. The chromosome solution was constructed as a two-part chromosome (Figure 2.18); the first part was the component sequencing solution and was modeled via a path representation. The second chromosome part was also modeled as a path representation, but in a different context. The second part represented the location of the component types in the feeder slots to be allocated. Thus, the first part of the chromosome would be much longer than the second part. If the AIM were modeled, the second chromosome would not be used at all.

Wong and Ji showed that the GA method is "easily adapted" towards many different types of assembly machines. This assertion was based on the fact that the only modeling change in their GA code for a machine, placing components on a specific board, would be the objective function subroutine and the presence (or absence) of a feeder rack in the case of an AIM verses either the PAPM or the RTHM. This work will be presented in greater detail as part of the solution approach in Chapter 3.

Chapter 3

Approach to Minimization of Time for Component Placement

The objective of this research is to develop a method by which panelization is used to improve the minimization of the component placement time. In order to create this approach, concise definitions the panelization parameters must be created. These parameters include those associated with the panel design, the PCB design, and the relationships between the two.

This chapter begins with mathematical formulations of the panel pattern generation and PCB rotations within those patterns. Once these definitions are established, the mathematical problem descriptions of the AIM, PAPM and RTHM will be developed. The terminology used to describe elements of the problem are defined in Table 3.1.

A *global approach* using GA will be described as the primary effort in analyzing the overall time minimization problem encompassing panelization and component placement and allocation parameters. The global approach encompassed all the decision variables in the problem description—panel patterns, PCB rotations, component placement sequence and component type allocation—and addresses all of these variables simultaneously. In order to have baselines against which to compare the global approach results, a second

approach is developed. This *two-stage approach* requires the development of a measure which can estimate the goodness of a particular panel design without solving the component sequencing and allocation problem. These measures are determined for all possible panel designs through complete enumeration. The panel design with the highest estimator value is then determined and used as the worst-case panel design. This worst-case design is then used in a traditional GA method to solve for the component placement and component allocation time minimization problem. The results of this traditional GA will be used for comparison against the global GA results. In the course of generating the estimated worst-case design, the best-case design can also be estimated for use in a traditional GA. This capability of selecting a proposed, single, best-case design, separate from the global GA, led to the creation of a second, best solution via the two-stage approach. These two approaches and the measures they generate will be discussed in Chapter 4.

3.1 Panel Pattern Generation

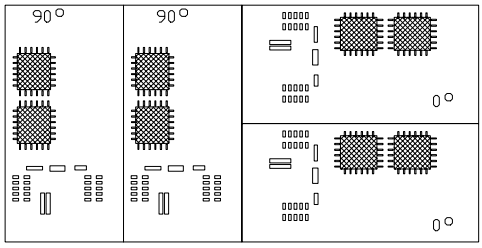
The creation of the panel pattern alternatives is necessary in creating the designs which may, in turn, potentially help reduce the component placement time. In the most restrictive sense, a panel with m PCB, all of which are identical in shape *and* circuit design (i.e., component locations), it is likely that multiple patterns are possible, even if rotations within the PCB boundaries are ignored. An example of such a situation was shown in Figure 2.2.

The pattern generation will be accomplished by modifying the additive method developed by Wang [79]. This method is straightforward and handles the following needs:

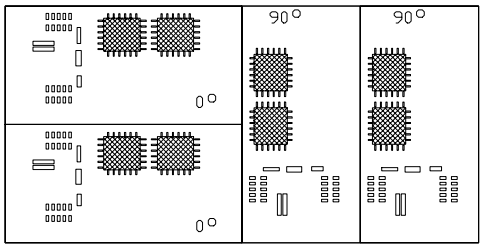
1. No repetition of patterns
2. Mirrored patterns (as required)
3. Allowance of nested subpatterns

Table 3.1: Panel pattern terminology.

H	Height of panel
W	Width of panel
m	Total number of PCB within panel
ν	Number of rectangle patterns in $L^{(M)}$
$L^{(\cdots)}$	Set of rectangular shape build patterns, or the pattern alternatives; $L^{(\kappa)}$ represents the set for the κ^{th} iteration $L^{(M)}$ represents the final set produced
$F^{(\cdots)}$	Set of all PCB which satisfy intermediate step requirements $F^{(\kappa)}$ represents the set for the κ^{th} iteration
R_i	A rectangle composed of sub-rectangle patterns; at iteration κ=0, all R_i are equivalent to the PCB_k singly
g_f	One of the final patterns in $L^{(M)}$; g_f contains all (X_{fk}, Y_{fk}, O_{fk}) for f, $k = 1, 2, \cdots, m$
X_{fk}	The center, horizontal location of PCB_k
Y_{fk}	The center, vertical location of PCB_k
O_{fk}	The orientations for each PCB_k within the R_i; for square PCB, only $O_{fk} = 0$ (for 0^o) is allowed; for rectangular PCB, $O_{fk} = 1$ (for 90^o) also allowed.
β_1	Maximum waste percentage allowable in interim R_i
b_k	Upper limit on the number of PCB_k in panel; for this problem, all $b_k = 1$, therefore b_k is not a factor here



a. Specific pattern for 4 identical PCB.



b. Mirrored pattern of shapes; circuit designs retained.

Figure 3.1: Mirrored PCB shapes result in different component locations for entire panel.

In cutting stock problems, a mirrored stock pattern has the same value when finding minimal waste or panel production solutions. In component placement problems, two mirrored patterns (PCB shapes, but not their circuit designs) will have different component locations for the entire panel, if the panel contains dissimilar PCB. An example of this potential problem is shown in Figure 3.1. The panel pattern shapes, or boundaries, of the PCB are mirrored from Figure 3.1a. and shown in Figure 3.1b. The circuit designs within those boundaries are not allowed to be mirrored; thus, the components have different spatial relationships with respect to each other. Component placement solutions may not be the same for the different layouts. And of course, more than the shown orientations for each of the PCB are possible within either of the patterns illustrated. One must be aware of these nuances if the additive method of panel generation, as given by Wang, is used in the panelization component placement problem [79].

Though items 1 and 2 are cited as advantages in the Wang method, the means by which the unwanted builds are avoided is not clear in the publication [79]. Apparently, these

duplications and reflections are generated as part of the method, then culled out through a comparison process. The generation of these reflections and duplicate patterns is due to the repetitive nature of the additive process. A subset of PCBs, or "build," is used over and over as the patterns are built up to the final shape. Groups of PCB can be mated together with their orders reversed, before the final patterns are established. This aspect is a feature which is actually desirable for pattern generation in the context of this research and the earlier discussions. Therefore, when the entire pattern (panel) orientation is important with respect to the global frame of reference (i.e., the machine frame reference), these reflected and rotated patterns need not be culled from the final solution set.

Within the restriction of requiring guillotine patterns with no waste, an algorithm for generating the panels may be developed (using definitions of Section 2.2):

Step 1. a.

b. Choose a value for $\beta_1, 0 \leq \beta_1 \leq 1$.

c. Define $L^{(0)} = F^{(0)} = \{R_1, R_2, \ldots, R_\nu\}$, and set $\kappa = 1$.

Step 2. a. Compute $F^{(\kappa)}$ which is the set of all rectangle T satisfying

(i) T is formed by a horizontal or vertical build of two rectangles from $L^{(\kappa-1)}$; the second addition of any build allowed to rotate $O_i = 0^o$ or 90^o before addition to the first rectangle,

(ii) the amount of trim waste in T does not exceed $\beta_1 HW$, and

(iii) those rectangles R_i appearing in T do not duplicate any PCB within the newly formed pattern.

b. Set $L^{(\kappa)} = L^{(\kappa-1)} \cup F^{(\kappa)}$. Remove any *identical* rectangle patterns from $L^{(\kappa)}$

Step 3. If $F^{(\kappa)}$ is nonempty, set $\kappa \longleftarrow \kappa + 1$ and go to Step 2. Otherwise,

Step 4. a. Set $M = \kappa - 1$.

b. Remove any patterns R_i that, when placed within the panel dimensions $H \times W$, result in trim waste.

c. If an AIM case problem, remove any *identical* rectangle patterns from $L^{(M)}$ which are identified from panel rotations (in 90^o increments) and compared to others in the set.

d. Order the rectangles (i.e., the patterns) of $L^{(M)}$ in a list, such that each of these patterns are represented by $L^{(M)} = \{g_1, g_2, \ldots, g_\nu\}$. Record the number of patterns, ν.

κ and ν were used in lieu of Wang's notation of k and n, respectively, to avoid confusion in the upcoming formulations involving component placement. Here, κ and ν refer to the iterations for Step 2 and the number of rectangle patterns in $L^{(\kappa)}$, respectively. No waste is allowed for any of the patterns in the final selection process, though some waste is permissible the interim rectangle combinations. Additionally, for the purposes of this research, each PCB is considered a unique initial R_i, even if more than one PCB is identical in shape to another. This requirement results in all of the bound constraints $b_1, b_2, \cdots, b_m = 1$. Another consequence of the last requirement is that each PCB_k may be defined in two-dimensional space by a unique (X_{fk}, Y_{fk}, O_{fk}). These parameters will be established as the centers and PCB orientation of each PCB_k within a pattern, for reasons which will become apparent in Section 3.2.

Rotations of the rectangular *shapes* are allowed in the vertical and horizontal builds; this option was mentioned as a possibility by Wang [79]. This option does not mean, however, that the rectangles are distinguishable beyond either 0^o or 90^o, since pattern formation is concerned only with the orientation of the boundary, and not any internal configuration (such as a circuit design). Squares would have no useful rotations at all, since their shape is identical for any right-angle orientation.

All of the patterns, combined, will be used as a resource for potential arrangement solutions to the component placement problem. Item 4c is added in the AIM case. AIM

problems have no external feeders, nor different component types; thus, any panel (pattern) rotation orientation inside such a machine would result in the same problem definition. However, as described earlier, the reflections of unique patterns are retained in the solution set, due to the PCB circuit designs (i.e., the PCB component locations).

With the set of all possible patterns generated for a set of panel dimensions, the next activity will be to develop a means by which to account for the PCB orientations within the panel and the component locations within each PCB.

3.2 Transformation of Rotated PCB Within Panels

Panelization is the common viewpoint to all of the machine types considered in this research. In this concept of the panel design, multiple PCB are arranged in a single panel. Each of the PCB has their own components defined relative to the individual PCB itself; thus, these components must be translated into the reference frame of the panel under consideration. For the mathematical description of this transformation, definitions are established in Table 3.2.

A generic PCB example is given in Figure 3.2. Let a PCB_k have a referenced orientation; that is, it has a center, (X_{fk}, Y_{fk}), and a rotation (in degrees) about that center, α_k. The rotation of that center is relative to the panel design, g_f, which was established in Section 3.1. Therefore, an α_k must be relative to the O_{fk} defined within the g_f. The m α_k for a given g_f is described as the PCB rotations. All possible PCB rotations which are feasible for g_f is described as the *PCB rotation set*, or $P^{(Q)}$, where the total members in this set is Q. The components are located via Cartesian coordinates, relative to the PCB center point. These coordinates, for the horizontal and vertical dimensions, are designated as $x_i^{''}$ and $y_i^{''}$, respectively. The horizontal and vertical dimensions of PCB_k are represented by l_k and w_k, respectively. Therefore, it is understood that the components have PCB coordinate ranges of $-\frac{1}{2}l_k \leq x_i^{''} \leq \frac{1}{2}l_k$ and $-\frac{1}{2}w_k \leq y_i^{''} \leq \frac{1}{2}w_k$.

Table 3.2: Transformation terminology.

H	Height of panel
W	Width of panel
m	Total number of PCB within panel
(X_{fk}, Y_{fk})	The center of each PCB_k, relative to panel coordinates.
$(x_i^{''}, y_i^{''})$	Coordinates of component i, relative to (X_{fk}, Y_{fk})
$(x_i^{'}, y_i^{'})$	Transformed component coordinates without PCB rotation
(x_i, y_i)	Transformed component coordinates including PCB rotation
α_k	PCB_k rotation about (X_{fk}, Y_{fk})
(l_k, w_k)	Horizontal and vertical dimensions of PCB_k, respectively
n_k	Number of component locations for PCB_k
N	$\sum_{k=1}^{m} n_k$
$\xi(i)$	Function for a given component i yields the corresponding PCB k
$a_{q\xi(i)}$	Binary decision variables
$\mathcal{L}_{fi}$	Orientation matrix for each component i in PCB_k; $O_f k = 0$
$\mathcal{L}_{fi}^{1}$	Orientation matrix for each component i in PCB_k; $O_f k = 1$
$P^{(Q)}$	Set of all possible PCB rotations which are feasible for any g_f in $L^{(M)}$.

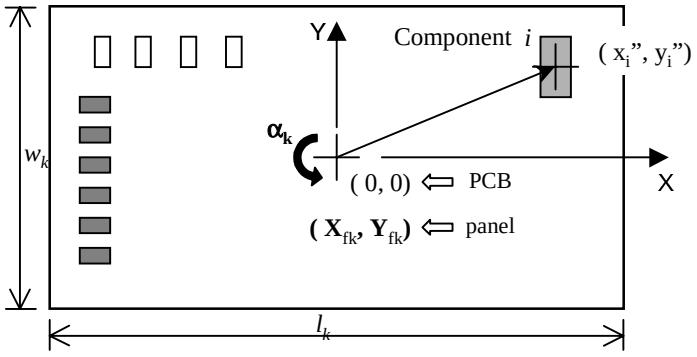


Figure 3.2: Local PCB Component Location at $O_{fk} = 0$.

A given panel under consideration (components omitted for clarity) has m PCB_k, as shown in Figure 3.3. An arbitrary, global reference point is taken as the lower, left-hand corner of the panel. The center of each PCB_k is located at panel (global) coordinates (X_{fk}, Y_{fk}). It should be noted that in Figure 3.3, PCB_2, PCB_3, PCB_4, and PCB_5 are square (i.e., $w_j = l_j$). Each PCB_k has n_k component locations, upon which the components will be mounted. A requirement in the problem formulation is that i is ordered sequentially to each consecutive panel. Thus, i for Figure 3.3 would be

$$i = \{1, 2, ..., n_1, n_1 + 1, ..., n_1 + n_2, n_1 + n_2 + 1, ..., \sum_{k=1}^{m} n_k\} \tag{3.1}$$

and $\sum_{k=1}^{m} n_k = N$. Also established is a function through which, for a given i, the corresponding k can be determined. This function is designated as $\xi(i) = k$.

If Figure 3.3 was the panelization pattern g_f for the panel, then the coordinates of an individual component i would be

$$\begin{aligned} x_i^{'} &= X_{fk} + x_i^{''} = X_{f\xi(i)} + x_i^{''} \\ y_i^{'} &= Y_{fk} + y_i^{''} = Y_{f\xi(i)} + y_i^{''} \end{aligned} \tag{3.2}$$

where $(x_i^{'}, y_i^{'})$ represent the transformed coordinates.

Within a pattern g_f [with all (X_{fk}, Y_{fk}, O_{fk}) established], one may alter the locations of the components without rearranging the location of the PCB centers. That is,

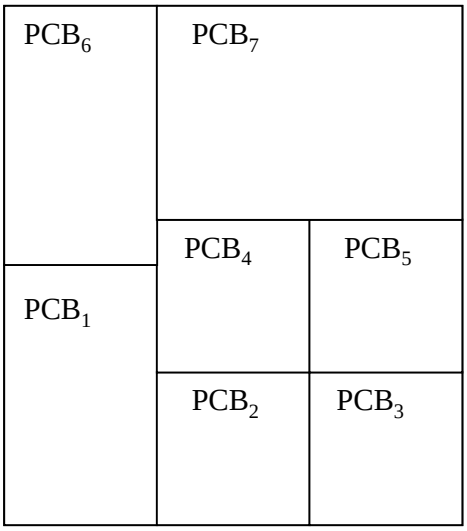


Figure 3.3: Panel Schematic.

rectangular PCB can be rotated in place by $\alpha_k = 180^o$. Square PCB can have rotations of $\alpha_k = 0^o, 90^o, 180^o$, or 270^o. The global transformed coordinates, including the rotation transformation due to α_k, are

$$\begin{aligned} x_i &= X_{f\xi(i)} + x_i^{''}\cos(\alpha_k) + y_i^{''}\sin(\alpha_k) \\ y_i &= Y_{f\xi(i)} + x_i^{''}\sin(\alpha_k) + y_i^{''}\cos(\alpha_k) \end{aligned} \tag{3.3}$$

It could be advantageous to avoid the trigonometric functions of equations 3.3. Since the rectangular shapes of the PCB limit rotations to 90^o increments, the cofactors of the PCB_k local coordinates take on values of 0, 1, or -1. Equations 3.3 can be rearranged so that binary decision variables will allow the appropriate cofactors to be engaged. Equations 3.3 can be rewritten as

$$\begin{aligned} x_i &= X_{f\xi(i)} + x_i^{''}a_{1\xi(i)} - x_i^{''}a_{2\xi(i)} + y_i^{''}a_{3\xi(i)} - y_i^{''}a_{4\xi(i)} \\ y_i &= Y_{f\xi(i)} + y_i^{''}a_{1\xi(i)} - y_i^{''}a_{2\xi(i)} - x_i^{''}a_{3\xi(i)} + x_i^{''}a_{4\xi(i)} \end{aligned} \tag{3.4}$$

where

$\alpha_k \Rightarrow$	0^o	180^o	90^o	270^o
$a_{1\xi(i)}$	1	0	0	0
$a_{2\xi(i)}$	0	1	0	0
$a_{3\xi(i)}$	0	0	1	0
$a_{4\xi(i)}$	0	0	0	1

$a_{q\xi(i)}$ are the binary decision variables. A single column of values for $a_{q\xi(i)}$ is allowed for each PCB_k, corresponding to a particular orientation, α_k. Note that for a rectangular PCB_k, $\alpha_k \neq 90^o$ or 270^o, thus requiring $a_{3\xi(i)} = a_{4\xi(i)} = 0$ for that k.

This arrangement allows a matrix form which is more compact:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} X_{f\xi(i)} \\ Y_{f\xi(i)} \end{pmatrix} + \begin{bmatrix} x_i'' & -x_i'' & y_i'' & -y_i'' \\ y_i'' & -y_i'' & -x_i'' & x_i'' \end{bmatrix} \begin{pmatrix} a_{1\xi(i)} \\ a_{2\xi(i)} \\ a_{3\xi(i)} \\ a_{4\xi(i)} \end{pmatrix} \tag{3.5}$$

Or, using matrix and vector notation,

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} X_{f\xi(i)} \\ Y_{f\xi(i)} \end{pmatrix} + [\mathcal{L}_{fi}]\, \vec{\mathbf{a}}_{\xi(i)} \tag{3.6}$$

$\mathcal{L}_{fi}$ is a matrix for each component i in PCB_k; the vector $\vec{\mathbf{a}}_{\xi(i)}$ takes on values corresponding to the column under the appropriate α_k in Table 3.2. If there are m PCB_k, and each PCB_k has n_k components, then there would be a total of $\sum_{k=1}^{m} n_k$ transformations. At this point, equations 3.3 through 3.6 have assumed that the shape orientations in g_f were all for $O_{fk} = 0$. If one of the PCB in Figure 3.3 was a shape for which $O_{fk} = 1$ in a particular g_f, then components in this PCB would have component local coordinates which shifted from their original definitions, such that $x_i'' \rightarrow y_i''$ and $y_i'' \rightarrow -x_i''$. This shift is illustrated in Figure

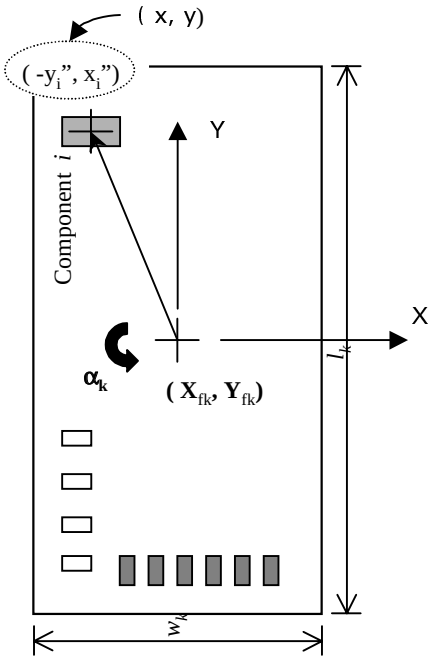


Figure 3.4: Local PCB Component Location for $O_{fk} = 1$.

3.4. For one of the PCB_k within a g_f where $O_{fk} = 1$, designate $\mathcal{L}^1_{fi}$ such that

$$\mathcal{L}^1_{fi} = \begin{bmatrix} y_i'' & -y_i'' & -x_i'' & x_i'' \\ -x_i'' & +x_i'' & -y_i'' & +y_i'' \end{bmatrix} \tag{3.7}$$

Equation 3.6 can be modified such that

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} X_{f\xi(i)} \\ Y_{f\xi(i)} \end{pmatrix} + \mathcal{L}_{fi}\vec{\mathbf{a}}_{\xi(i)}(1 - O_{fk}) + \mathcal{L}^1_{fi}\vec{\mathbf{a}}_{\xi(i)}O_{fk} \tag{3.8}$$

Before panelization, (x_i, y_i) were constants in the component placement problem. With the inclusion of variable patterns and rotations within those patterns, (x_i, y_i) become functions which require the set parameters from g_f and values for $\vec{\mathbf{a}}_{\xi(i)}$ available within that pattern set.

If a PCB_k is rectangular, $\vec{\mathbf{a}}_{\xi(i)}$ is constrained to take on only those values under the first ($\alpha_k = 0^o$) and second ($\alpha_k = 180^o$) columns. The PCB_k could be classified as belonging to set R if the shape is rectangular and not in the set if they are square.

Vector $\vec{\mathbf{a}}_{\xi(i)}$ of equation 3.8 has constraints of

$$\sum_{q=1}^{4} a_{q\xi(i)} = 1 \qquad \forall i \tag{3.9}$$

$$a_{3\xi(i)} + a_{4\xi(i)} = 0 \qquad \forall \xi(i) = k \text{ in which } k \in \mathbf{R} \tag{3.10}$$

$$a_{q\xi(i)} = 1,\ 0 \qquad \text{binary}$$

Equation 3.9 results in N constraints, though all the equations for each k are identical. Thus, in reality, these constraints are only m in number. Equation 3.10 results in only up to m constraints as well, depending on the number of rectangular PCB. Another way to represent equation 3.10 is to simply have $a_{2\xi(i)} = a_{4\xi(i)} = 0$ for those PCB_k in which $j \in \mathbf{R}$.

3.2.1 Nomenclature for Addressing PCB Rotations

The problem addressed above can be cumbersome to address in a mathematical manner, so a shorthand method of describing a set of PCBs and their rotations is defined. A pattern containing m PCB_k must be said to have a *PCB rotation set* which describes the rotation of each PCB, relative to the pattern's PCB initial orientations. As described above, a rectangular PCB will take on rotational values of $\alpha = \{0^o, 180^o\}$. Since there are only two values available, they can be represented in binary, such that 0 and 1 correspond to 0^o and 180^o, respectively.

A square PCB will take on rotational values of $\alpha = \{0^o, 90^o, 180^o, 270^o\}$. These four values may be represented as a quadral set, such that 0, 1, 2 and 3 correspond to 0^o, 90^o, 180^o and 270^o, respectively. Therefore, if a panel consisted of a rectangular PCB_0 and a square PCB_1 , then a PCB rotation set with $PCB_0 = 90^o$ and $PCB_1 = 270^o$ can be represented as `(13)`. This nomenclature assumes that the set is read from left to right, with the first PCB rotation starting from the left. This nomenclature also allows for an easy means of determining the total number of PCB rotation sets available for a given pattern alternative.

The total number of PCB rotation sets is defined as

$$2^{(\text{number of square PCB}\times 2)+(\text{number of rectangular PCB})} \tag{3.11}$$

This representation is appropriate in view of the fact that the number of square PCB rotations possible can be represented as the enumeration of two binary digits.

3.3 Objective Functions for Component Placement with Panelization

As shown in Chapter 2, much of the component placement problems have been formulated in an IP format. Panelization characteristics will be introduced into these objective functions to illustrate the increased solution space created through this consideration.

3.3.1 Automatic Insertion Machine

The distance matrix d_{ij} defined in equation 2.6 is no longer valid when considering panelization, since the component locations, x_i, y_j, are no longer static in the panel design for component i. Instead of a two-dimensional distance matrix, one may consider a four-dimensional distance matrix d_{ijfp}. The added dimensions come from the definitions of $L^{(M)}$ and $P^{(Q)}$, the pattern alternatives and PCB rotation sets, respectively. Thus, the new distance matrix will be of size $N \times N \times Q \times M$.

The objective function in equation 3.12 should now be altered to be

$$\textit{Minimize:} \quad \sum_{f\in L^{(M)}} \sum_{p\in P^{(Q)}} \sum_{i=1}^{N} \sum_{j=1}^{N} d_{ijfp} u_{ijfp} \tag{3.1}$$

(3.1

Table 3.3: AIM formulation terminology.

m	Total number of PCB within panel
(x_i, y_i)	Transformed component coordinates including PCB rotation
N	Total number of components on the panel
$L^{(M)}$	Set of all pattern alternatives for the panel
$P^{(Q)}$	Set of all possible PCB rotations which are feasible for any g_f in $L^{(M)}$.
d_{ijfp}	Distance between component i and component j
u_{ijfp}	Binary decision variable: *true* (=1) if component i immediately precedes placement of component j for pattern alternative f and PCB rotations p; *false* (= 0) otherwise

be calculated as

$$d_{ijfp} = \sqrt{(x_{ifp} - x_{jfp})^2 + (y_{ifp} - y_{jfp})^2} \tag{3.14}$$

Enumerating this distance matrix is computationally possible. However, additional constraints must be added to the traditional TSP formulation (not covered here). The problem remains that the formerly NP-hard TSP problem now has added variables in the context of f and p. Thus, the added complexity of panelization would result in an NP-hard formulation as well. Consequently, heuristic methods will be developed in order to obtain solutions in a reasonable amount of time.

3.3.2 Pick and Place Machine

Without consideration of panelization, the PAPM problem formulation differs from the AIM problem primarily in the addition of feeder locations and component types. The distance matrix is composed of the distances between each component on the panel and each of the available feeder slots on the machine (and off the panel edges). Kumar and Li give a problem

Table 3.4: PAPM formulation terminology.

$\overline{N}$	Maximum number of component types on panel
$\mathcal{S}$	Number of slots available in feeder rack
v_{hl}	Binary decision variable: *true* (=1) if feeder slot h contains component type l and *false* (= 0) otherwise
$\tau(i)$	Function for a given component i yields the corresponding type l

definition and formulation of all the constraints in [42]; the objective function is

$$\textit{Minimize:} \quad \sum_{i=1}^{N}\sum_{h=1}^{\mathcal{S}}\sum_{j=1}^{N}(d_{ih}+d_{jh})u_{ij}v_{h\tau(j)} \tag{3.15}$$

The parameter $\tau(j)$ is a translation function which provides the component type l for the component j. v_{hl} a binary decision variable where it is *true* (=1) if feeder slot h is assigned component type l and *false* (= 0) otherwise. In this case, there are two distance metrics, representing a two-part motion of the placement head in the following sequence:

$$(x_i, y_i) \implies (x_h^{\tau(j)}, y_h^{\tau(j)}) \implies (x_j, y_j)$$

d_{ih} is the distance between component i and the slot h which contains component type $\tau(j) = l$. This component type is that needed for the subsequent component j placement. The next distance in the sequence is d_{jh}, which is the distance between the aforementioned slot h for $\tau(j) = l$ and the component j placement location. The paths d_{jh} and d_{ij} are the required motions and nonrequired motions, respectively, as defined by Ball and Magazine [3].

When panelization is considered in the objective function, the distance matrix becomes four-dimensional, in that the additional two dimensions are formed due to the definitions of the pattern alternative and the PCB rotation sets. The objective function can be

written as

$$\textit{Minimize:} \quad \sum_{f \in L^{(M)}} \sum_{p \in P^{(Q)}} \sum_{i=1}^{N} \sum_{h=1}^{\mathcal{S}} \sum_{j=1}^{N} (d_{ihfp} + d_{jhfp}) u_{ijfp} v_{h\tau(j)} \tag{3.16}$$

The summations required for the pattern alternatives and PCB rotations are the same as for equation 3.15, as well as the binary decision variable u_{ijfp} for the placement sequence. However, the $v_{h\tau(j)}$ decision variable is not associated with any panel design selections (which involve $L^{(M)}$ and $P^{(Q)}$).

3.3.3 Rotary Turret Head Machine

As noted in Chapter 2, the RTHM is similar to the PAPM in that a feeder is a resource for multiple component types. However, the RTHM has all three moving subsystems: A rotary head (no linear $x-y$ motion), an $x-y$ moving table upon which the panel is mounted, and a linearly moving feeder bank. For purposes of illustration, the bank is defined as moving back and forth in the x direction. Figure 2.10 is repeated in Figure 3.5 for convenience in this section.

The three moving subsystems must coordinate for the pick and the place actions on the panel. The placement always occurs at the same, global location in the machine; the pick up also has a fixed location, but on the opposite periphery of the turret. The ideal circumstance is for the placement on the panel to occur at the same time the pick up will occur on the other side. Also, each subsystem does not move at the same rate; therefore, formulation must be in terms of time, not distances. The RTHM is designed with the anticipation that the turret will be the quickest to complete its actions between component placements [84, 16, 15]. In the traditional RTHM concept, the rotary head is described as "indexing" to the next nozzle (which holds the subsequent component to be placed). The turret head is said to have a tooling "gap," between the placement and the pick up component, which is half of the number of placement nozzles available on the head (assuming an even number of nozzles). For convenience, the gap is designated as δ. In Figure

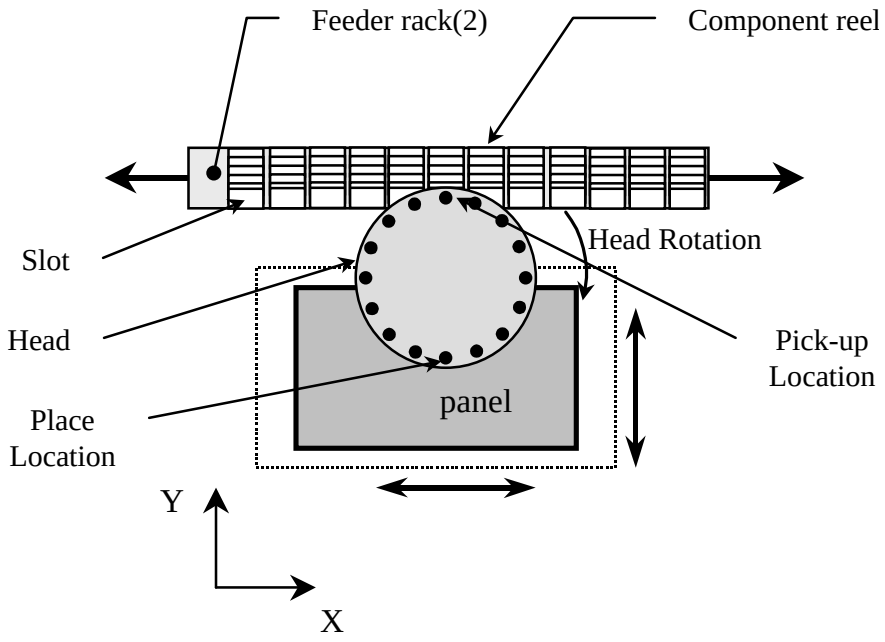


Figure 3.5: Rotary turret head machine (RTHM) schematic.

3.5, the δ is eight.

The slowest of the three subsystems will determine the time required between placement of component i and component j. Thus, the placement time of the entire panel is again dependent upon a specific component placement sequence and component feeder allocation. Let the traveling time between component i and j placements for the table, the feeder bank and turret indexing be $t_1(i,j)$, $t_2(i,j)$ and $t_3(i,j)$, respectively. If the turret indexes at the same rotary speed during the carry of any component, then $t_3(i,j)$ may simply be designated as t_3. It is assumed that all turret nozzles are filled and released during panel placement (i.e., no sequential skipping of nozzles during pickup or placement).

A new translation function $\phi(l)$ yields a slot h for the component type l. $(x_{\phi(\tau(i))}, y_{\phi(\tau(i))})$ is a means by which the slot location can be found for a specific component i.

The table travel distance from component i to component j placement locations is

Table 3.5: RTHM formulation terminology.

$\phi(l)$	Function for a given component type l yields the corresponding slot h
$(x_{\phi(\tau(i))}, y_{\phi(\tau(i))})$	Location of slot $\phi(l) = h$ which holds components of type $\tau(i) = l$
δ	The gap between the pick up and placement, in terms of turret nozzles
$t_1(i,j)$	Table motion time between component i and component j locations
$t_2(i,j)$	Time required between pick up of component i to that of component j
t_3	Time required for turret index from component i to component j
$\mathbf{t}_i$	Time required for placement of a component i

given in equation 2.6. The time required for table travel is therefore

$$t_1(i,j) = \frac{d_{ij}}{v_1}$$

where v_1 is the constant velocity of the table between component i and j placement locations.

In the case of the feeder motion, one should note that the pick up for a component, i, is occurring (sequentially) $\delta - 1$ components earlier than the component being placed at the same time. With that concept in mind, the time required for the feeder motion between the two slot locations for component i placed immediately before j would be

$$t_2(i,j) = \frac{\sqrt{(x_{\phi(\tau(i))} - x_{\phi(\tau(j))})^2 + (y_{\phi(\tau(i))} - y_{\phi(\tau(j))})^2}}{v_2} \tag{3.17}$$

where v_2 is the constant velocity of the feeder bank between slots occupied by $\tau(i)$ and $\tau(i)$.

The index time between component i and j would be simply t_3, assuming each component requires the same speed of the turret during its carry on the turret.

Therefore, the placement time, $\mathbf{t_i}$, for a single component i on the panel would be

$$\mathbf{t_i} = \max\{t_1(i-1,i), t_2(\tau(i+\delta-1), \tau(i+\delta)), t_3\}$$

If the turret has the ability to carry multiple components during placement, then at the start of a batch of like panels, the first panel would be serviced by a turret which would be completely empty of components when the placement sequence began. The assumption will be made that the turret will fill itself to at least one less than half its capacity before the panel is seated for placement operations to begin. For the example in Figure 3.5, the first 7 components in the placement sequence will have already been loaded into the turret nozzles before the placement cycle begins. This practice can be extended to assume that every subsequent panel will have the last $\delta - 1$ placements require the pick up of the first $\delta - 1$ components in the panel placement sequence.

With the preloading assumption for batch production, the RTHM objective function is the summation of the $\mathbf{t_i}$:

$$\textit{Minimize:} \quad \sum_{i=1}^{\mathbf{N}} \mathbf{t_i} \tag{3.18}$$

When panelization is considered, the selection of the pattern alternative and PCB rotation set must be accounted for. The distance matrix, d_{ijfp} can be once again considered as a four-dimensional matrix; in this situation, it is identical to that of the AIM and can be constructed with equation 3.14. The time required to traverse a distance d_{ijfp} would be then

$$t_1(i, j, k, p) = \frac{d_{ijfp}}{v_1}$$

Similar to the PAPM panelization formulations, the feeder motion time between successive components is independent of the panel design; thus, this time represented by equation 3.17 is essentially unchanged. The constant head index time is unchanged in the panelization considerations as well. Thus, the time for component i to be placed in the context of a panelized component placement solution would be

$$\mathbf{t_{ifp}} = \max\{t_1(i-1, i, f, p), t_2(\tau(i+\delta-1), \tau(i+\delta)), t_3\} \tag{3.19}$$

An RTHM objective function which would account for panelization could take on the form

$$\textit{Minimize:} \quad \sum_{f \in L^{(M)}} \sum_{p \in P^{(Q)}} \sum_{i=1}^{\mathbf{N}} \mathbf{t_{ifp}} \tag{3.20}$$

Equation 3.20 is not linear at all, in light of the consideration that equation 3.18 was not linear before the additional complexity associated with panelization. This objective function is useful in computational search techniques, however, and will be the basis for obtaining RTHM solution scores in the genetic algorithm techniques developed in this research.

3.4 The Genetic Algorithm Method

The introduction of panelization to the component placement problem increases its complexity. As stated in Chapters 1 and 2, the probabilistic, directed, search method, GA, will be used for this problem. It has the advantages of evaluating the search space simultaneously (in each GA iteration) for all the decision variables. It also will present a convenient means of representing the solution in terms of a chromosome description. GA has also been used successfully for a variety of component placement machines, though without panelization aspects [48, 49, 83].

As an example, consider the very simple, 3-PCB layout as shown in Figure 3.6. There are only three PCB in the panel, but each has a different, internal component layout (circuit design). Thus, though PCB_2 and PCB_3 have the same, square dimensions, swapping the location of each PCB in a pattern will result in a different panel component layout. Also, the machine under consideration has a single feeder bank; thus, mirrored patterns will produce different component pattern presentations to the machine for component placement calculations. Note that if an AIM were under consideration (no feeder rack), then only patterns g_1 and g_2 would be required. There are eight different patterns necessary for consideration. Also, possible within each pattern are different orientations of the PCB_k. With one rect-

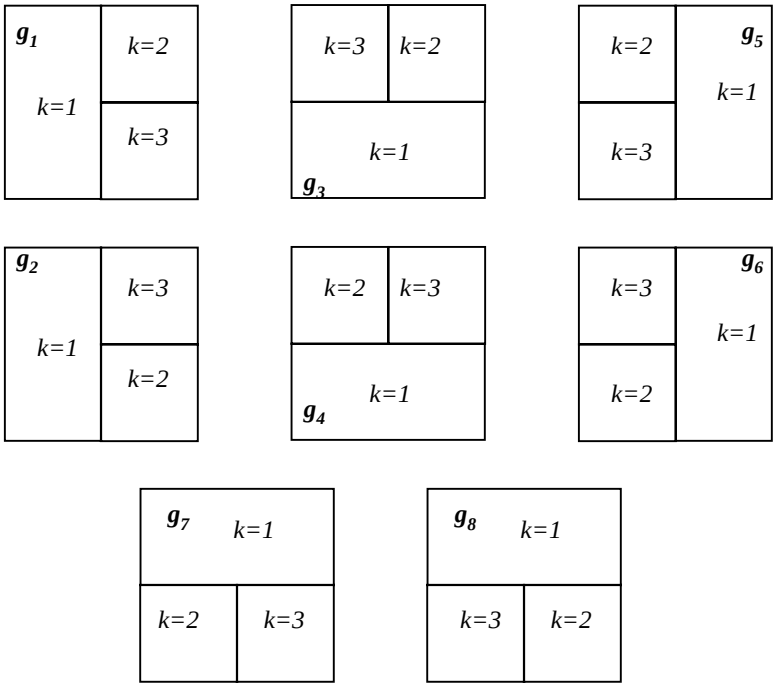


Figure 3.6: Panel example.

angle and two different squares, equation 3.11 would yield $2^{(2)(2)+1} = 32$ different PCB_k rotation combinations within a specific pattern g_f, producing a total of $8 \times 32 = 256$ panel configurations to investigate within the machine. Such a number of different problems to solve may be daunting for a manufacturing engineer to accomplish singly upon his existing placement machine software. Thus, needed is a combinatorial technique that reduces the placement time by optimizing over the entire design space. The GA technique will be used as such a technique; its application to this problem is described in the next section.

3.4.1 Genetic Algorithm Procedure

The GA will be used in a steady-state population scenario. Convergence and maximum iteration limits will be combined to end the process. The following lists the GA steps:

Step 1. Initialize the problem.

a. Randomly generate Q chromosomes for the initial population, S_O.

b. Set counters

i) gen=1.

ii) ϱ=1.

c. Establish the current generation, S_Q, as S_O.

d. Evaluate fitness **F** for S_Q.

Step 2. Select individuals from S_Q and place into mating pool MP_Q (Section 3.4.4).

Step 3. Apply genetic operators, G_j, to mating pool at their respective rates, r_j.

a. Record which individual offspring were produced by which genetic operators, G_j

b. Collect offspring into the offspring pool, MP_O.

Step 4. Evaluate **F** for MP_O individuals.

Step 5. $MP_Q \cup MP_O = MP_U$, a large, temporary pool of last generation and offspring.

Step 6. Keep fittest Q individuals in MP_U.

a. Sort the MP_U individuals in inverse fitness order.

b. Define new S_Q as first Q individuals in MP_U.

c. Record which G_j produced surviving offspring.

d. Set $\varrho = \varrho + 1$

Step 7. Check termination criteria

a. If $gen = TC_1$, then the maximum generations criteria is met.

i) Stop iterations and record top chromosome solution.

ii) Else,

b. If minimum convergence termination criteria, TC_2, is met,

i) Stop iterations and record top chromosome solution.

ii) Else,

Step 8. If ϱ=*Threshold*,

a. Readjust r_j to new adaptive rates (Section 3.4.5)

b. Set $\varrho = 1$

Step 9. $gen = gen + 1$

Step 10. Return to Step 2

Details of the above process will be discussed next.

3.4.2 Solution Representation

In this document, the chromosome contains the solution information to the entire problem; the term "individual" can also represent the chromosome when discussing populations. An individual i is represented symbolically as $\tilde{s}_i$. For convenience, a chromosome will be represented symbolically as $\tilde{s}$. The gene j, or $\tilde{s}(i)$, is a single piece of information within the chromosome, from which a smaller portion of the chromosome is not possible. Each $\tilde{s}$ is composed of sections, or "links;" a link is a string of genes which provide complete information for a portion of a problem. This set of links can also be referred to as a "composite" chromosome.

Depending upon the machine type, a chromosome which represents the solution may consist of three to four separate links. Table 3.6 illustrates an example of a component placement solution for a panel with characteristics of four PCB ($m = 4$), eight components on the panel ($N = 8$), and three component types ($\overline{N} = 3$) in three feeder slots ($S = 3$).

The first link, $\tilde{s}^{\rho}$, will contain a single gene which represents the identification f of the g_f pattern for the panel. The second link, $\tilde{s}^{\theta}$, will contain m genes. Each $\tilde{s}^{\theta}(k)$ corresponds

Table 3.6: Layout of chromosome links.

$\tilde{s}^{\rho}$	$\tilde{s}^{\theta}$				$\tilde{s}^{\sigma}$								$\tilde{s}^{*}$		
f of g_f $\Downarrow$	α_k $k \Downarrow$				sequence of component i $order \Downarrow$								allocation of component type l to slot h $h \Downarrow$		
	1	*2*	*3*	*4*	*1–2–3–4–5–6–7–8*								*1*	*2*	*3*
3	0^o	0^o	90^o	270^o	3	1	8	5	2	7	4	6	2	1	3
$\uparrow f \uparrow$	$\uparrow \alpha_k \uparrow$				$\uparrow i \uparrow$								$\uparrow l \uparrow$		

to the rotation of the PCB_k within pattern g_f. The value of the genes within these links is conceptually defined in terms of degrees rotation, but in computational terms, they will be mapped from sets of four, binary values representing a_{qk} for each PCB_k.

Links $\tilde{s}^{\rho}$ and $\tilde{s}^{\theta}$ could be considered the PCB orientation map, since one link could not exist in the solution without the other. For this reason, a single vertical line is drawn between these links in Table 3.6 to signify a connection between them.

The third link, $\tilde{s}^{\sigma}$, represents the component placement sequence. Each gene's position in the link is the order of placement for the components, when the link is read from left to right. The component identification, i is the integer value in the gene.

The fourth link, $\tilde{s}^{*}$, shows the feeder slots allocation of component types. Each gene's position in the link represents the slot designation, h. The component type identification, l is the integer value in the gene. In the case of the AIM, the third link will not be used, since component types are not an issue in such a problem.

3.4.3 Fitness Functions

The fitness functions are those distance and time functions in Sections 3.3.1 through 3.3.3. Of interest is the fact that the decision variables in those problem formulations are not explicitly incorporated into the GA, since the sampling process establishes those decisions within each chromosome. For reference, the fitness function is designated as $\mathbf{F}$ and the fitness of a particular chromosome, $\tilde{s}$, is $\mathbf{F}\{\tilde{s}\}$.

3.4.4 Selection Operation

The roulette wheel method, as introduced in Section 2.4.2, will be used to assign probabilities of selection to the mating population. The probability, p_i, for each chromosome i, is defined by:

$$p_i = \frac{\mathbf{F}_i}{\sum_{j=1}^{Q} \mathbf{F}_j} \tag{3.21}$$

Where $\mathbf{F}_i$ is the fitness of the chromosome and Q is the population size. The cumulative probability, P_i is then

$$P_i = \sum_{j=1}^{i} p_j \tag{3.22}$$

For each successive generation, Q individuals are copied into the mating pool by this process. There is a possibility that multiples of the same individual may be copied into the pool, but this is an allowable feature of the GA process.

Once the mating pool is established, genetic operators act upon the mating pool with their prescribed rates.

3.4.5 Genetic Operators

A genetic operator, G_j, is the function which generates new links from the mating pool individuals. A total of $Z = 5$ genetic operators are applied to the pool at operation rates of

Table 3.7: Genetic Operators.

Description	$\tilde{s}^{\rho}$	$\tilde{s}^{\theta}$	$\tilde{s}^{\sigma}$	$\tilde{s}^{*}$
G_1:Mutation (std)	$\surd$			
G_2:Mutation (swap)		$\surd$	$\surd$	$\surd$
G_3:OX			$\surd$	$\surd$
G_4:Inversion			$\surd$	$\surd$
G_5:Rotation			$\surd$	$\surd$

r_j, where r_j is the ratio of offspring produced relative to the number of individuals in the mating pool. If the mating pool population is Q, and the desired offspring produced from the G_j is o_j, then the rate should be established at $r_j = o_j/Q$. To produce Q offspring from all the operators in each generation, $\sum r_j = 1.0$.

There will be five G_j used in the GA method developed as part of this research The operators are summarized in Table 3.7 as to their applicability on particular links in the chromosome.

Mutation is the procedure where an individual in the mating pool has the value of a random gene changed. There are two mutation operators, the standard, G_1, and the swap, G_2. The standard mutation changes the current value within the gene to a new, feasible value in a random manner. For a binary value, the change is simple: $1 \rightarrow 0$ and $0 \rightarrow 1$. For values in a restrictive range, such as for $\tilde{s}^{\theta}$, the random selection must be chosen from one of the two (four) possible values for the rectangular (square) PCB represented in the gene.

The swap mutation operator, G_2, was developed by [83] for path representation chromosomes. In these chromosomes, a random change in one gene to a different value, even to a feasible, new value, would produce a subtour situation (in TSP terms) and result in the omission of a required data point. As an example, consider a mutation of a 5-component $\tilde{s}^{\sigma}$ parent link (i.e., in the mating pool). Mutation of a single value in the 3rd gene would pro-

duce a duplication of the placement of component 3 and omission of placement of component 5:

1	3	**5**	2	4	6	⇒	1	3	**3**	2	4	6

The swap mutation randomly selects a *pair* of genes within the link, in order to change their respective values and still retain a valid path representation.

1	3	**5**	2	**4**	6	⇒	1	3	**4**	2	**5**	6

The OX crossover operator, G_3, was originally developed by Oliver [62]. This operator was illustrated in section2.4.3. Its purpose is to transport a portion of one parent link path information to the other, without creating subtours.

The inversion operator, G_4, reverses the order of a segment within a single, parental link [49]:

1	**3**	**5**	**2**	**4**	6	⇒	1	**4**	**2**	**5**	**3**	6

The rotation operator, G_5, selects a segment within a link, positions the last gene value to the first in that segment, and shifts the rest of the gene values to the right by one gene position [49]:

1	**3**	**5**	**2**	**4**	6	⇒	1	**4**	**3**	**5**	**2**	6

Adaptive Genetic Operator Rates

The adaptive GA allows r_j to be adjusting after each ϱ generations (or iterations). This method was used by Wong and Leu to aid convergence of their component placement problems [83]. A tally of the number of surviving offspring, GT_j, which were produced by specific G_j is kept up to that generation. A new set of rates, r'_j is established by normalizing the respective ratios of the tallies:

$$r_j^{'} = \frac{\sum_{i=1}^{\varrho}(o_j)_i}{\sum_{j=1}^{\mathbf{Z}}\sum_{i=1}^{\varrho} o_j} \tag{3.23}$$

If any of the $r'_j < 1/Q$, it would be likely that the G_j would not produce, on average, any new offspring for the next generation. At the next ϱ generation, the r_j may possibly be set to zero. In order to avoid this possibility, any $r'_j < 1/Q$ is set to $r'_j = 1/Q$ and $o_j = \varrho Q$. Then equation 3.23 is used to reevaluate the r'_j and $r_j = r'_j$.

3.4.6 Termination Criteria

There will be two termination criteria used in the GA. The first is the maximum generations threshold, TC_1. When the GA produces TC_1 generations, the GA will stop and the best solution will be reported.

The second termination criteria, TC_2, is the convergence *threshold*. The best solution from an earlier number of generations is stored. After the current generation has produced offspring and the new S_Q is formed, the ratio of the earlier, stored solution and the current solution is calculated. If that value is less than the convergence ratio, TC_2, then the GA is halted. The TC_2 is equivalent to the slope of the improving solutions over the GA generations.

3.4.7 Initialization

The GA is initialized by generating an initial population of chromosomes, S_0. The initial population is created via random component sequence and allocation generation, which is a common method of initializing genetic algorithm populations [83, 48, 49, 17, 55].

3.5 The Estimator Functions

A basic assumption in this research is that the potential for improvement in the component placement time can be achieved if panelization were to be considered in the panel design. The term "potential" is used because of the possibility that the best panel design may have been selected in a traditional approach as a matter of course. This situation is very likely for panels with only a few PCB. For example, Figure 3.7 illustrates all possible PCB rotation sets for a simple pair of PCB to be used in an AIM. Note that there is only *one* pattern possible for this situation, since the AIM TSP problem solution does not require an orientation of the pattern relative to the machine. The person in charge of creating the panel array of these two PCB may likely be inclined to arrange both PCB in the same orientation relative to each other, as shown in Figure 3.7 (A). This panel design, along with (D), can be seen by inspection to be the best pattern design for this problem, since the groups of components are closest to each other. The rationale behind the "by inspection" declaration is manifest by the concept that the shortest distances between the two component groups (i.e., each PCB) can be defined in sequence for each PCB. It is noted also that in this special case, the (A) and (D) designs are identical with respect to the AIM machine situation.

The reason for this illustration is to highlight the fact that a traditional analysis using designs (A) or (D) should produce the best possible outcome with respect to cycle time reduction considerations. If the global GA heuristic were conducted which would include all four designs of Figure 3.7, the best possible outcome should result in nearly the same value as that for a traditional analysis on designs (A) or (D). Thus, a measure of the potential for component placement time improvement would be to compare the traditional analysis on the worst pattern possible for panelization against the best possible pattern. The global GA should produce results which will be at or near the best traditional analysis results.

In considering the above, it should be noted that to truly find the best and worst possible panelization patterns, one must actually enumerate through all the possible panel designs, which are composed of the PCB shape patterns and PCB rotation sets for each. As

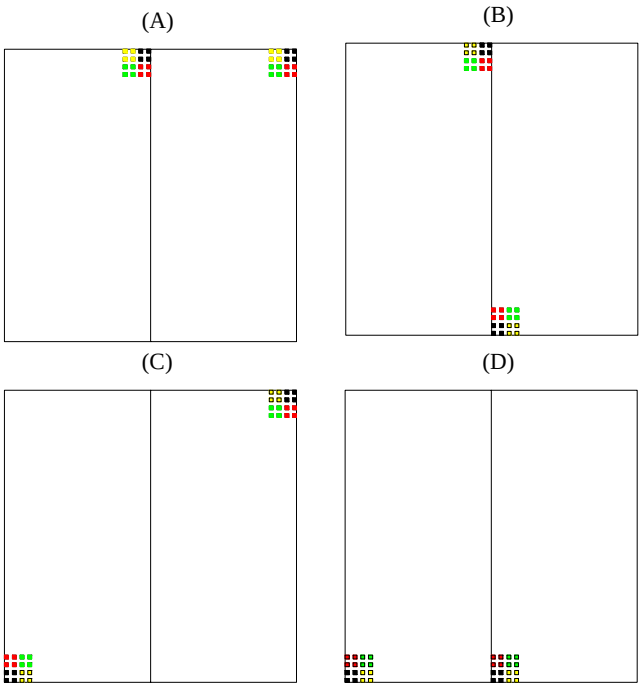


Figure 3.7: Simple example of two PCB in a panel design.

suggested in section 3.3.1, the component placement solution process for fixed PCB in one pattern is NP-hard; heuristics (in this case, the GA) are used to handle this traditional problem [48, 49, 17]. To perform the GA on all the alternative patterns for problems larger than two PCB can become time-prohibitive. An approximate measure to address the potential of good solutions is desired which will allow for enumeration of the panel design alternatives without requiring the time-consuming analysis of the complete GA upon the component sequencing and allocation problem.

A simple estimator function can be established for each machine type. Each machine type will be addressed below and the estimator function defined for each. Symbols will be used from Tables 3.2 and 3.3, though additional ones will be defined below as necessary.

3.5.1 AIM Estimator Function

The AIM problem has been presented as a TSP. If the locations are allowed to shift, then one observation is that the closer the groups of locations are to each other, the more likely the TSP solution algorithm will select improved solutions. In the case of panelization, groups of components are allowed to shift their positions. These groups could be identified in terms of their PCB component distribution centers, represented by (x_k^{CP}, y_k^{CP}) in rectilinear coordinates. For a given PCB_k, these centers are defined as

$$(x_k^{CP}, y_k^{CP}) = \left(\frac{\sum_{i=1}^{n_k} x_i}{n_k}, \frac{\sum_{i=1}^{n_k} y_i}{n_k} \right) \tag{3.24}$$

The center of the panel's entire component distribution, (X^{CP}, Y^{CP}), would be defined as

$$(X^{CP}, Y^{CP}) = \left(\frac{\sum_{i=1}^{m} x_k^{CP}}{m}, \frac{\sum_{i=1}^{m} y_k^{CP}}{m} \right) \tag{3.25}$$

For the entire panel, the objective would be to reduce the distances between all of these component groups. The simplest means of accomplishing this objective would be to sum the magnitudes of the differences between each PCB component population center and the panel component population center:

$$\text{EF(AIM)} = \sum_{k=1}^{m} \left(\sqrt{(X^{CP} - x_k^{CP})^2 + (Y^{CP} - y_k^{CP})^2} \right) \tag{3.26}$$

This value can be considered as the AIM estimator for the potential of component placement time improvement through the consideration of panelization. A graphical presentation of the EF(AIM) is presented in Figure 3.8 (A). The largest centroid symbol represents the center of the aggregate, panel component distribution and the smaller centroid symbols represent the two centers of the PCB component populations. The sum of the magnitude for the lines connecting these symbols is the value for the EF(AIM).

3.5.2 PAPM Estimator Function

The PAPM problem was presented in Section 3.3.2. The placement head must essentially make two paths for each component placement, these paths having been describe as the

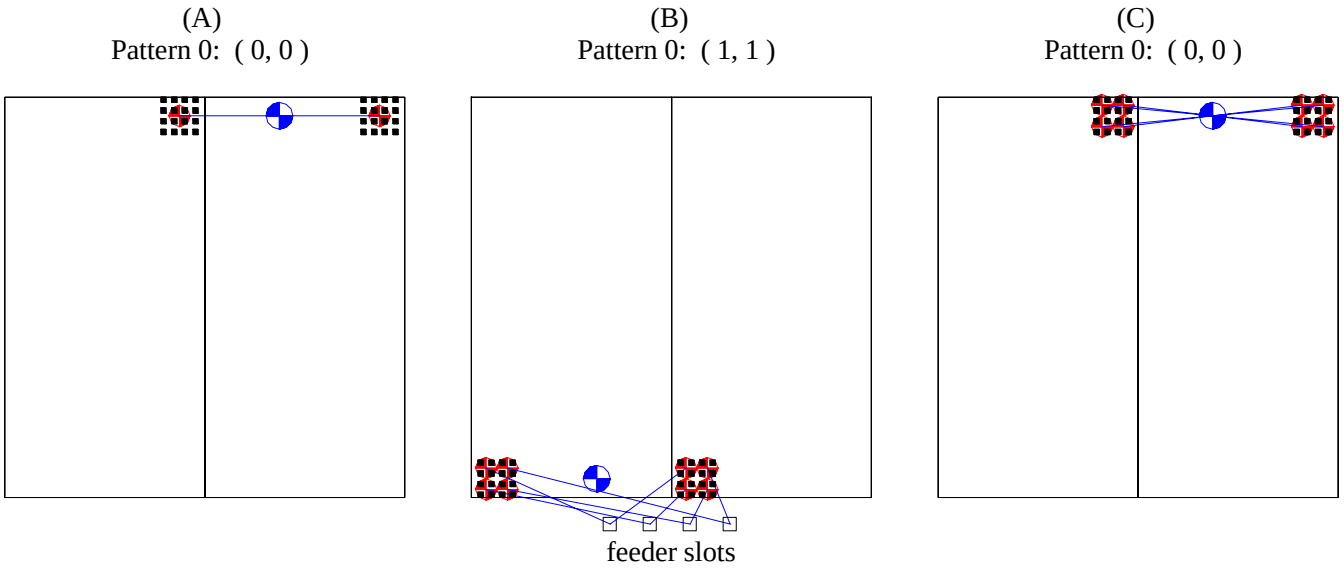


Figure 3.8: Graphical illustration of EF(AIM), EF(PAPM) and E(RTHM).

required and unrequired moves. Therefore, the AIM estimator formulation would not be appropriate in the PAPM model. From a visual perspective, the patterns with groups of components closest to the feeder locations would stand a better chance of resulting in low placement solution times. The PAPM has an additional element of its construction: That of component allocation. The (x_k^{CP}, y_k^{CP}) locations do not contain any information as to the component types within their associated distributions. A formulation for this situation is needed to represent each of the groups of component types on each PCB. This definition is the center of PCB k, component type j, distribution or, $(x_{kj}^{CTP}, y_{kj}^{CTP})$. These coordinates are based upon component types as well as locations within a PCB design and can be defined as

$$(x_{kj}^{CTP}, y_{kj}^{CTP}) = \left(\frac{\sum_{j=1}^{S} \sum_{i=1}^{n_k} x_i c_{ij}}{n_k}, \frac{\sum_{j=1}^{S} \sum_{i=1}^{n_k} y_i c_{ij}}{n_k} \right) \tag{3.27}$$

where c_{ij} is 1 if component i is of component type j and 0 otherwise. The total number of component types and the number of component types on each PCB_k are S and n_k, respectively. Once these subgroup centers have been established, a quick method by which to allocate their types to appropriate feeder slots is necessary. The largest candidate rule

(section 2.3) is used to form an initial assignment; a pair-wise exchange method is used to improve the measure: The total *weighted* distance between each $(x_{kj}^{CTP}, y_{kj}^{CTP})$ and its assigned feeder slot. The weighted distance is

$$\text{EF(PAPM)} = \frac{\sum_{j=1}^{S} \sum_{k=1}^{m} n_{kj}^{CT} \sqrt{(x_{kj}^{CTP} - x_j^{slot})^2 + (y_{kj}^{CTP} - y_j^{slot})^2}}{N} \tag{3.28}$$

where n_{kj}^{CT} represents the number of components of type j on PCB k. PCB component groups with a greater number of components for a given component type need to be weighted more than others of like component types with less components. Therefore, the PAPM estimator is a distance weighted by the individual PCB component type populations. The division of the estimator value by N is not strictly necessary for the problem, but does allow the estimator value to remain in the same units as the EF(AIM).

A graphical presentation of the EF(PAPM) is presented in Figure 3.8 (B). In this case, the magnitudes for the lines between the centroid symbols and the feeder slot locations must be scaled by the number of component types in each group, then divided by the total component population of the panel. Thus, the figure is illustrative, but cannot be used to directly measure the EF(PAPM).

3.5.3 RTHM Estimator Function

For the previous two machine types, the estimator functions have had the appeal of being a measure which is applied to the panel design without having to solve the component placement solution. This characteristic allows for fast calculation of these measures, and thus allows for complete enumeration of all panel design alternatives without an onerous delay. The RTHM problem has both the TSP and the component allocation elements of the previous two machine types. However, it is difficult to integrate the competing aspects of table travel, head index , and feeder travel times into a single measure without considering specific solutions for the entire component placement problem.

It can be noted that the panel design does distinguish between component type lo-

cations by nature of the component locations in themselves; therefore, the $(x_{kj}^{CTP}, y_{kj}^{CTP})$ coordinates could be used in some capacity to represent the TSP aspect of the problem. In general, if the distances between all of these component groups were reduced, the impact of table travel time could be reduced in consideration; in such a reduction, the remaining competing factors (head index and feeder travel times) would be a consequence of a particular solution. Therefore, a simple measure for the RTHM panel design would be to sum the magnitudes of the distances from the center of the panel component distribution to each $(x_{kj}^{CTP}, y_{kj}^{CTP})$ (as opposed to the more general method for the AIM). The center of the panel component population, (X^{CP}, Y^{CP}), was defined in equation 3.27; to form the estimator for EF(RTHM), equation 3.28 must be modified to account for the distinction of component types

$$\text{EF(RTHM)} = \frac{\sum_{j=1}^{S} \sum_{k=1}^{m} n_{kj}^{CT} \sqrt{(x_{kj}^{CTP} - X^{CP})^2 + (y_{kj}^{CTP} - Y^{CP})^2}}{N} \tag{3.29}$$

A graphical presentation of the EF(RTHM) is presented in Figure 3.8 (C). In this figure, the magnitudes for the lines between the smaller centroid symbols and the larger, centroid symbol must be scaled by the number of component types in each group, then divided by the total component population of the panel.

Given the above estimating measures, one can now conduct panelization experiments on a series of panel design cases without having to solve the true component placement sequences and allocations for each panel design alternative. The panel design with the lowest estimator value can be used as a worst-case panel design scenario for the component placement problem and compared against the global approach to the problem. Since all enumerations of the panel designs are examined, it is a simple matter to save the best and worst estimator-valued panel design during the enumeration process. This best-valued panel design can then be used in a traditional GA approach to solve the component placement problem as well; the results can then be compared against the global approach to assess ability of either method to reach similar, low scores.

Estimator function limitations

It should be noted that the estimator formulations *do not* mathematically prove a panel design will either 1) be the absolute optimal design from which the minimum placement time will be derived or even 2) the panel design selected is the single best design for the heuristic used (the GA, in this case). The estimator values are merely point-valued indicators which designate some panel designs as having groups of components which *may* lead to better (and worse) total solutions than others.

The global GA approach was developed in section 3.4; this approach is used to determine a best panel design and component placement solution. However, a panel design from the estimator functions can also be used in a traditional GA approach to produce a potentially best panel design. These two solutions may result in either the same final component placement time or the same panel design. Of the two different approaches, it is anticipated that the estimator scores should correlate to the magnitudes of the corresponding traditional GA scores produced from their designated designs. It is also anticipated that the global GA scores and panel designs will correlate to the estimator scores, but there is no guarantee. Experiments will be designed in Chapter 4 to explore these proposals.

Chapter 4

Experimental Design

This chapter presents the experimental design and related computational issues. First presented is a justification of the PCB and panel parameters used throughout most of the experiments. The subsequent section addresses computational issues associated with running the experiments and the GA in the context of panelization. Next, factors common to most of the experimental designs are established. The chapter closes with details of each of the five major experimental designs.

There are 5 experiments:

- *Experiment 1 (E1):* Verification of the panelization GA program.
- *Experiment 2 (E2):* Tightly grouped, increasingly eccentric component PCB locations.
- *Experiment 3 (E3):* Uniformly spaced component locations with eccentric component type designations.
- *Experiment 4 (E4):* A panel and circuit design from industry.
- *Experiment 5 (E5):* Randomly generated PCB component layouts.

E1 is a verification of the GA computer program developed in this research; it is used to

compare its results against known problems for the three machine types (AIM, PAPM and RTHM). Experiments E2 and E3 use predetermined PCB layouts in order to address different component location and component type distributions. Experiment E4 uses a panel and circuit design from the PCCA industry. The last experiment, E5, uses random component locations in the PCB designs. Each of these experiments are described in sections 4.5, 4.6, 4.7 and 4.8.

4.1 Justification of Experimental Parameters

Three of the experiments which dealt directly with panelization (E2, E3, and E5) used three panel design scenarios: two-PCB, four-PCB, and eight-PCB. There were no academic publications found to support these panel PCB population figures; however, several trade journals do discuss up to eight PCB within a panel [58, 66, 39]. Thus, the eight-PCB alternative is chosen as the upper value for an experimental case. Based upon the boundary established in section 4.1 below, the four-PCB alternative was selected in order to have square PCB shapes available for experiments with identical PCB in a square panel. The two-PCB alternative is used as the minimum number of PCB required in order to have a panelization scenario.

Most PCB designs are complex and are composed of many components and component types. In making such generalizations, it would be advantageous to specify more tightly the parameters of PCB and panels. Thus, a survey was conducted of the literature in component placement research to determine the range of values for electronic panel dimensions, component population sizes, types and other parameters. Figures 4.1 through 4.3 show the results of the survey. There were two major disadvantages with the publications reviewed:

1. The authors often illustrated only very simple examples; such examples were acknowledged as not representative of the industry norm.

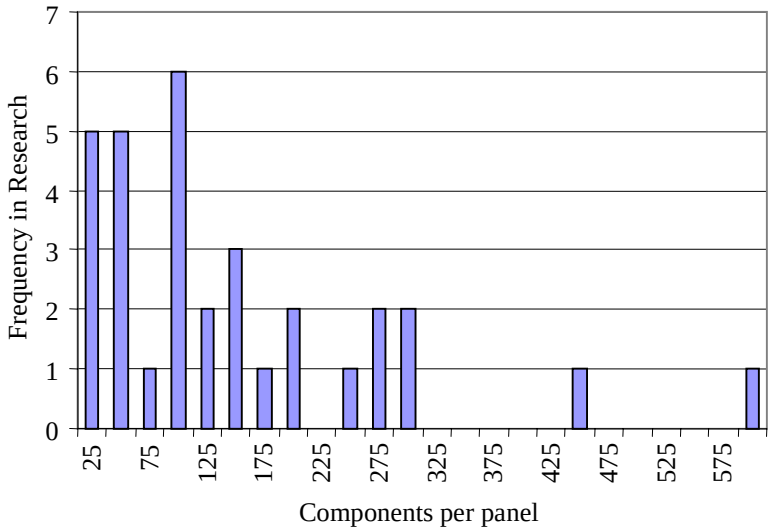


Figure 4.1: Histogram of panel component populations in literature.

2. Researchers frequently left unspecified many panel dimensions, component population numbers, machine speeds and other parameters.

In spite of the above difficulties, the survey did yield some information of interest. These charts show no readily apparent component layout distributions; therefore, the figures were used as a basis for parameter ranges in the experimental designs. Of immediate interest is the panel width and height. Based upon Figure 4.1, a 300 by 300 millimeter boundary would be reasonable for the panel dimensions, as it falls within the range of chart values. This panel boundary will remain the same for all experiments but the industrial design example. The panel boundary restriction forces the PCB under consideration to be of different dimensions, depending upon the number of PCB in the panel layout.

For the predetermined PCB component layouts experiments (E2 and E3), it was desired to fall within the general ranges of Figure 4.1. Sixteen components per PCB were used for these designs, such that the smallest component population size would be 32 for the two-PCB situation and 128 for the situation PCB case. This range of total component populations falls within that of the survey illustrated in Figure 4.1.

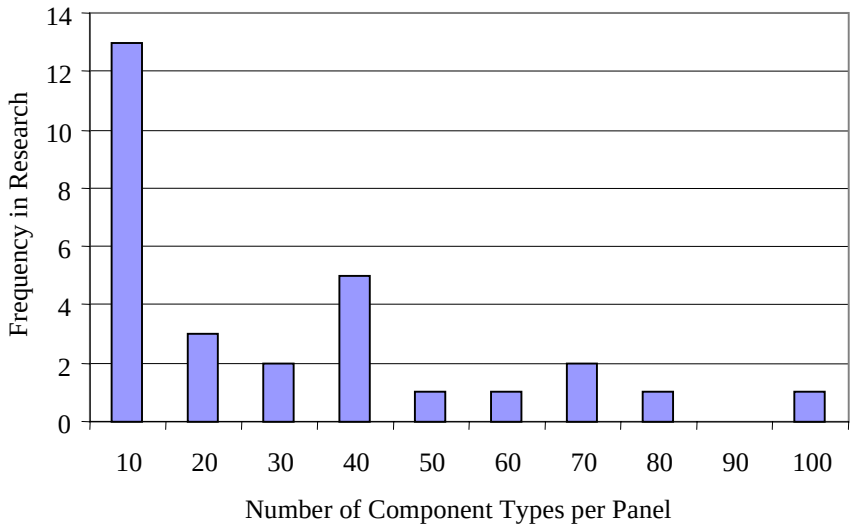


Figure 4.2: Histogram of panel component type populations in literature.

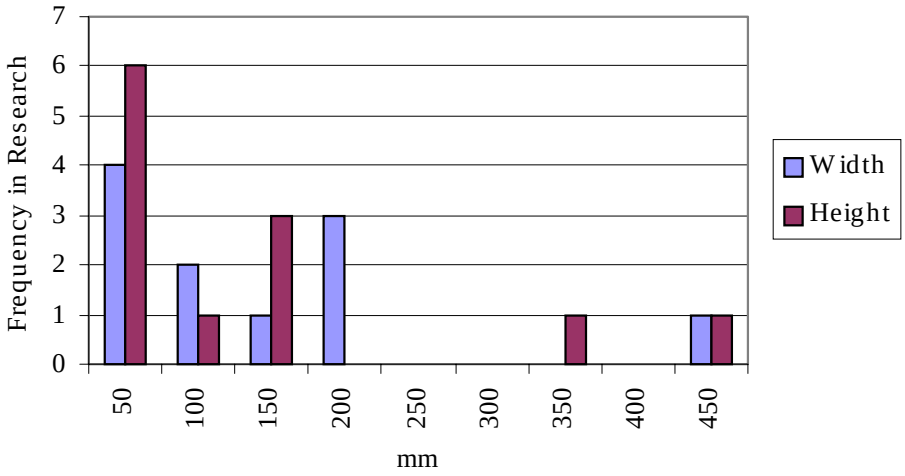


Figure 4.3: Histogram of panel dimensions in literature.

This survey also was referenced for the PCB random design experiment (E5), but a set range for the PCB component population (from 100 to 150 components per panel) was used for each machine type/panel scenario.

Of note is the fact that when there are eight identical PCB under study, the experiments require exploration of multiple patterns in addition to the PCB rotation sets within each pattern. Even for the simpler AIM machine, there will be 12 patterns investigated (Figure 4.4). The PAPM machine would have a total of 36 patterns possible for the 8 PCB situation; these pattern alternatives are displayed in Appendix F.

The AIM and PAPM can have total distance traveled by the placement head as the measure for experiments, assuming a constant head (or table) velocity. If a velocity value were assumed, one need only divide by that value to obtain an estimated cycle time in seconds. However, the RTHM requires the index time, feeder bank velocity, and table velocity in order to determine the cycle time for the PCB. The literature produced only scattered information on such parameters. Therefore, for the experiments in this dissertation, these values will be taken from one of the more complete RTHM examples by DeSouza and Lijin, where 0.15 seconds, 133.3 millimeters/second, and 133.3 millimeters/second are the head index time, feeder bank velocity, and head (or table) velocity, respectively[16].

It should be noted that the RTHM placement algorithm performance can be highly dependent upon the machine parameters themselves. In the published literature, the cases examined are almost always devoted to a specific machine which has certain index, table and feeder motion speeds. With a distance between adjacent feeder slots of 0.15 millimeters, the time the feeder travels between adjacent slots is 0.15 seconds, which matches the index time. If the table speed is sufficiently fast with respect to the distance between any two components, such that any component-to-component motion will be less than the index timing, then that table motion is no longer a part of consideration to the problem. Therefore, the locations of the components on the panel no longer become an issue. This observation is extended to the panelization issue, since it impacts the RTHM solutions from the TSP perspective. In these

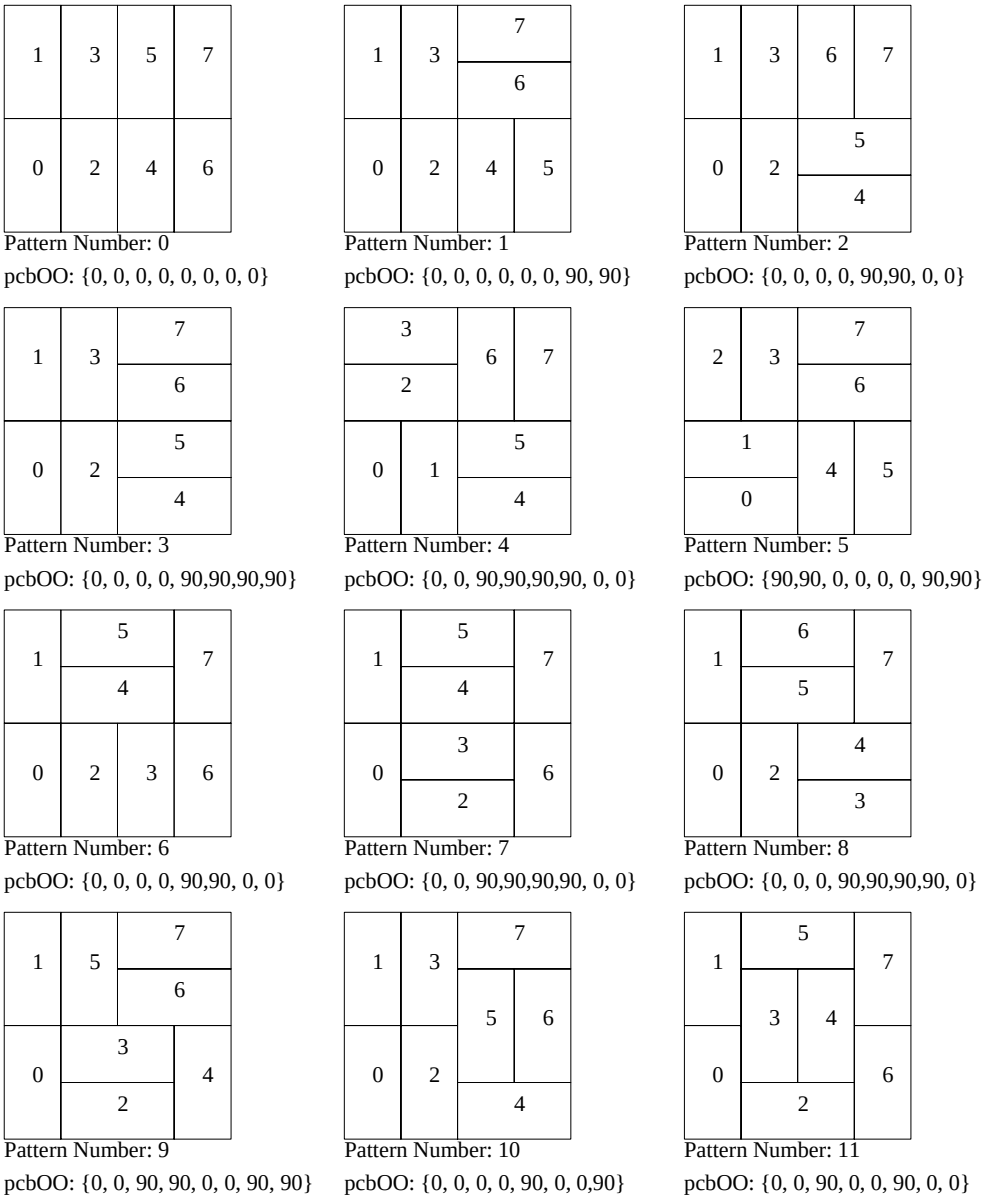


Figure 4.4: Pattern alternatives for the AIM type, 8-PCB experiments.

experiments, any path traveled by the table which takes less than 0.15 seconds will eliminate the TSP portion of the problem; this distance corresponds to about 20 millimeters. On the other hand, the component types and their respective population numbers will always be the same, regardless of the panelization design.

4.2 Computer Programs and Computational Issues

A program which includes all of the panelization factors described in this chapter was developed to handle a large variety of cases. The C++ program is listed in Appendix A. This program, designated as `panelizer`, can perform the GA solution procedure for all three machine types: AIM, PAPM, and RTHM. Additionally, it has the ability to either read in PCB component locations and types (i.e., the PCB designs) or it can generate random PCB design sets. The program is controlled via command-line arguments, but also requires data input files. The software and its operation are described in detail in Appendix A.

A second program, `pmaker`, was also developed to generate patterns from given PCB shapes and is based upon the algorithm developed in section 3.1. The output from `pmaker` is used as an input data file for `panelizer`; the code is provided in Appendix B.

Another program, `finder`, was developed to generate quick estimator values of all pattern and PCB rotation possibilities, in order to select the best and worst candidates for a traditional GA approach. This estimator value is a result of the formulations described in section 3.5. Only the best and worst panel designs were retained; in case of ties, the first discovered extreme would be kept. The program is addressed in Appendix C.

The underlying improvement algorithm is the GA. The solution produced from such an algorithm has some dependency upon the random number seed used in the computations. GA are typically performed on the same problem several times but with different random number seeds. Then the best solution of the repeated program runs is used as the final value. In the examples to follow, three repetitions were performed for each experimental

instance; only the lowest score for each experimental instance is reported and compared in this dissertation.

4.2.1 GA Implementation in Distance-related Problems

There is a computational time disadvantage when panelization is incorporated into the GA computational procedures. The TSP problem can be addressed by establishing a matrix of all possible distances between a given component and all the others. This distance matrix needs to be calculated only once for a panel which does not have the panelization aspect to consider. The TSP calculating algorithm will then select appropriate, static values from this matrix during its operation. This technique is computationally efficient, since the multiplications and square-root calculations required for distance calculations are performed only at the start of a solution algorithm; additions are then performed for each candidate solutions. Addition-based calculations are much faster than multiplication calculations; therefore, repeated multiplications and square-root calculations are to be avoided (and are mostly avoided in this situation) [63, 44].

However, when considering panelization in a distance-related problem, one must account for having PCB which (along with their components) can move to different locations within a panel's boundaries as well as these PCB being able to rotate their orientation. This added complexity means that a single, static distance matrix is no longer available for the GA, since the components may present themselves in a new position during the optimization process. As an example: A computer implementation of a TSP solution technique must recalculate the distance matrix every time a new PCB orientation or panel pattern is presented. The fundamental practical problem with this approach is that machine calculation time is correspondingly increased. Compounding this problem is that the greater the number of components per panel, the greater their impact upon the distance matrix recalculation time penalty.

The `panelizer` program (Appendix A), since being developed for use in panelization

experiments, uses the distance recalculation method for the global analysis situation. For a small AIM panelization problem of 30 components, the problem takes only two minutes to run through 4000 generations. However, more realistic problems involving more than 100 components can require an hour to run through the same number of generations on a Sun Ultra 1 workstation, using the global analysis technique. Thus, the convergence termination option described in section 3.4.6 is used for the experiments to reduce the time required to obtain a solution. Additionally, an option in `panelizer` allows for the traditional approach cases to be run with a static distance matrix for appropriate problems. This option reduces the solution time for those problems as well.

4.2.2 Output Nomenclature

A panel design, as described in Chapter 3, consists of a pattern (or layout) of PCB shapes and a PCB rotation set allowable within that pattern. In the course of representing such designs, a shorthand method is used in this dissertation. The pattern set is generated, from which a pattern can be referenced by a single number. As an example, the patterns available for an eight-PCB panel in an AIM may be referenced with any number from 0 through 11, representing the 12 patterns possible for that problem (Figure 4.4). Each pattern has a feasible set of PCB rotations, depending upon the PCB shape. A rectangle, restricted in a pattern, may be in only one of two rotational positions: 0^o or 90^o. The eight rectangular PCB of Figure 4.4 may be represented by a feasible rotation set of {0,0,0,0,1,1,1,1}, where the first four PCB are rotated 0^o and the second four PCB are rotated 0^o. If there were eight *square* PCB in a panel, then each of the feasible PCB rotations, 0^o, 90^o, 180^o, and 270^o could be represented by the numbers 0, 1, 2, and 3, respectively. Experimental results will be presented in this shorthand method in order to save space in the output graphics and tables.

4.3 General Experimental Design

The terms *global GA* and *global approach* indicate that the component placement/allocation problem was addressed by including all patterns, PCB rotation sets, component sequencing and component types allocation as variables in the problem. The global approach uses all of these independent variables in the GA, altering them as specified in section 3.4.1 in the course of obtaining a solution. This type of approach is contrasted against the *traditional approach*, wherein the panel design is not alterable during the GA process; only the component sequencing and component types allocation parameters are allowed as variables in the component placement minimization problem, the assumption being that the panel design has been previously determined by means other than the GA. In the course of this research effort, the panel design for the traditional approach is determined through the application of the estimator functions as described in section 3.5. The traditional approach is similar to the method used in the research publications to this date, with the caveat that panelization parameters are not addressed, since they are assumed as fixed before the component placement and allocation analysis is begun.

The experimental designs for the research are tabulated in Tables 4.1, 4.2, 4.3, 4.4 and 4.5. These tables show the progression of the experiments and the subgroups of experiments within the larger experimental layout. These subgroups are associated with the machine types and panel scenarios (i.e., PCB per panel) appropriate for the experiments. E1 does not share the format of the other experiments, since it does not address panelization.

E2 and E5 will display results for all three machine types. E3 only addresses two: The PAPM and RTHM. For E2, E3, and E5, within each of the machine types, panel design alternatives will be examined for two, four and eight-PCB layouts. E1 is strictly the computer program verification experiments, where panelization is not examined; thus, the PCB layout alternatives are not appropriate for that experimental set.

E2 has a special situation where there are a series of three different panel designs

examined separately for each machine type/panel layout scenario. The rationale for this experimental approach is discussed in section 4.5.

For each experimental instance, E2, E3, E4, and E5 will have five different output results. The first two results, *Estimated Worst* (EW) and *Estimated Best* (EB), are from the estimator functions (section 3.5) applied to particular machine type and panel layout at hand. These two results are predictions of which designs will have the worst and best placement times in the experimental instance under consideration. The next pair of results are called the *Traditional Worst* (TW) and *Traditional Best* (TB) results; they are the results of the traditional analysis performed separately upon the worst and best panel designs as predicted by the Estimated Worst and Best results, produced earlier. The fifth result for each experimental instance is from the global analysis, where no panel design was specified before starting the GA approach.

The majority of the experiments (E2, E3, E4, and E5) are conducted to determine whether there are differences in the placement times for different panel designs, given the same machine type, PCB component location/type layout, and PCB per panel. It is anticipated that the Traditional Worst design will produce placement time solutions which should be greater than those produced from either the Traditional Best design or the global analysis-produced design. This difference should be anticipated as the *largest potential for improvement* (LPI) in the placement time solutions, given the machine type, PCB design, and panel scenario (PCB per panel) under consideration. The LPI is an indication of how much improvement in the panel cycle time is possible if the panel designer chose the best panel design with respect to choosing the worst one.

Table 4.1: Experiment E1: Verification.

MACH	*Baseline*	*Research Results*
AIM	*Established in [48]*	Verified
PAPM	*Established in [48]*	Verified
RTHM	*Established in [48]*	Verified

Table 4.2: Experiment E2: Eccentric component group locations.

MACH	*No. of PCB*	*CASE*	*MACH*	*No. of PCB*	*CASE*	*MACH*	*No. of PCB*	*CASE*
AIM	2	A	PAPM	2	A	RTHM	2	A
		B			B			B
		C			C			C
	4	A		4	A		4	A
		B			B			B
		C			C			C
	8	A		8	A		8	A
		B			B			B
		C			C			C

Table 4.3: Experiment E3: Component type eccentricity.

MACH	*No. of PCB*
PAPM	2
	4
	8
RTHM	2
	4
	8

Table 4.4: Experiment E4: Industry PCB/panel design example.

MACH	*No. of PCB*
PAPM	8
RTHM	8

Table 4.5: Experiment E5: Random PCB designs ("*No. of PCB*" displayed horizontally to facilitate page layout).

	No. of PCB 2	*No. of PCB* 4	*No. of PCB* 8
MACH	Random Samples	Random Samples	Random Samples
AIM	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
	8	8	8
	9	9	9
	10	10	10
PAPM	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
	8	8	8
	9	9	9
	10	10	10
RTHM	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
	8	8	8
	9	9	9
	10	10	10

There are two different procedures which can be used to produce "best" panel designs: 1) The global analysis, which is always conducted to produce a theoretically best design, and 2) a traditional analysis performed on an Estimated Best panel design. Since these two heuristic procedures can produce two different results, there are two LPI measures possible. The Traditional LPI (TLPI) is based upon the difference between the Traditional Best and the Traditional Worst results and can be expressed either as a measurement difference value or as a percentage, such that

$$TLPI = TW - TB \qquad \text{or} \qquad TLPI = \frac{TW - TB}{TW} \times 100 \tag{4.1}$$

The Global LPI (GLPI) is based upon the difference between the global analysis and the Traditional Worst results and can be expressed either as a measurement value or as a percentage, such that

$$GLPI = TW - G \qquad \text{or} \qquad GLPI = \frac{TW - G}{TW} \times 100 \tag{4.2}$$

The estimator functions (equations 3.26, 3.28 and 3.29) yield the Estimated Best and Worst designs in a computationally short period of time and are independent of the component placement minimization process under consideration (in this research, the GA). The percent difference between the Estimated Best and Estimated Worst scores are reported along with the TLPI and GLPI. This score, the Estimated LPI (ELPI), is calculated as a percentage, such that

$$ELPI = \frac{EW - EB}{EW} \times 100 \tag{4.3}$$

The ELPI numerical value *does not* represent a prediction of what a traditional or global analysis score would produce in terms of the LPI; rather, the ELPI is compared against ELPI for other panel design alternatives in the same machine type/panel layout scenario for a relative ranking of yielding the largest potential improvement solutions. However, the ELPI will also be examined to see if there is a correlation between itself and the particular LPI being calculated. The ELPI, TLPI and GLPI will be reported for experiments 2, 3, 4 and 5.

4.4 Experiment 1

The results of E1 are used to check the construction of the GA program (`panelizer`) against published examples. The GA program is run against three cases in literature which have in common the implementation of genetic algorithms, component placement, component allocation, and the three machine types AIM, PAPM, and RTHM. These examples were published by Leu et. al.; however they do not include any panelization aspects [48].

The first verification example is an AIM situation; as such, there are no component types in this problem. The component locations are presented in Table 4.6 with the coordinates relative to the panel's lower, left-hand corner. The placement solution for the AIM will be in terms of distance (millimeters) traveled by the placement head. This distance is directly related to the total placement time for a constant-velocity assumption.

The PAPM problem requires both the solution of the component placement sequence as well as the component allocation to the feeder slots. The component locations and types are presented in Table 4.7 with the coordinates relative to the panel's lower, left-hand corner. The placement solution will be in terms of distance (millimeters) traveled by the placement head. This distance is directly related to the total placement time for a constant-velocity assumption.

Table 4.6: AIM component locations [48].

cmpnt#	**X**	**Y**	cmpnt#	**X**	**Y**	cmpnt#	**X**	**Y**
0	16.00	18.10	10	78.30	41.20	20	60.70	18.10
1	8.30	41.20	11	77.50	35.60	21	9.70	26.80
2	4.40	16.00	12	13.50	40.80	22	28.70	24.00
3	81.00	41.20	13	86.10	28.90	23	3.50	35.80
4	1.80	23.80	14	87.10	41.20	24	7.80	35.60
5	5.70	41.20	15	17.60	29.80	25	36.30	35.00
6	11.20	35.60	16	40.10	35.50	26	20.80	29.40
7	55.20	41.20	17	32.30	41.20	27	74.80	41.80
8	80.70	22.70	18	84.20	41.20	28	86.40	22.30
9	63.90	27.30	19	34.30	18.10	29	17.70	24.00

Table 4.7: PAPM component locations (X, Y) and types (CT) for verification example [48].

CT	**X**	**Y**	**CT**	**X**	**Y**	**CT**	**X**	**Y**	**CT**	**X**	**Y**	**CT**	**X**	**Y**
4	100	40	3	140	40	5	180	40	7	220	40	1	100	300
1	100	50	8	140	50	2	180	50	1	220	50	1	100	310
7	100	60	1	140	60	9	180	60	7	220	60	0	100	320
2	100	70	4	140	70	4	180	70	8	220	70	8	100	330
0	100	80	6	140	80	6	180	80	1	220	80	7	100	340
7	100	90	7	140	90	4	180	90	0	220	90	8	100	350
3	100	100	0	140	100	3	180	100	4	220	100	7	100	360
5	100	110	2	140	110	9	180	110	5	220	110	2	100	370
9	100	120	5	140	120	1	180	120	4	220	120	2	100	380
0	100	130	4	140	130	2	180	130	4	220	130	1	100	390
3	100	140	7	140	140	7	180	140	1	220	140	3	130	300
3	100	150	3	140	150	3	180	150	8	220	150	2	130	310
1	100	160	8	140	160	3	180	160	9	220	160	5	130	320
4	100	170	5	140	170	1	180	170	7	220	170	6	130	330
0	100	180	6	140	180	0	180	180	9	220	180	1	130	340
4	100	190	3	140	190	7	180	190	3	220	190	0	130	350
4	100	200	5	140	200	4	180	200	6	220	200	5	130	360
6	100	210	5	140	210	5	180	210	5	220	210	1	130	370
6	100	220	2	140	220	0	180	220	4	220	220	2	130	380
8	100	230	5	140	230	2	180	230	5	220	230	6	130	390
5	120	40	4	160	40	4	200	40	3	240	40	2	160	300
5	120	50	5	160	50	4	200	50	0	240	50	7	160	310
7	120	60	8	160	60	8	200	60	8	240	60	1	160	320
6	120	70	8	160	70	8	200	70	6	240	70	1	160	330
1	120	80	5	160	80	7	200	80	3	240	80	2	160	340
1	120	90	4	160	90	7	200	90	8	240	90	8	160	350
6	120	100	4	160	100	8	200	100	1	240	100	2	160	360
9	120	110	4	160	110	6	200	110	2	240	110	5	160	370
5	120	120	8	160	120	0	200	120	5	240	120	8	160	380
2	120	130	1	160	130	0	200	130	7	240	130	0	160	390
6	120	140	3	160	140	5	200	140	4	240	140	3	190	300
2	120	150	9	160	150	7	200	150	9	240	150	7	190	310
7	120	160	1	160	160	0	200	160	0	240	160	3	190	320
9	120	170	3	160	170	5	200	170	2	240	170	7	190	330
7	120	180	6	160	180	8	200	180	0	240	180	7	190	340
1	120	190	7	160	190	8	200	190	4	240	190	6	190	350
0	120	200	7	160	200	5	200	200	3	240	200	2	190	360
2	120	210	1	160	210	2	200	210	5	240	210	3	190	370
4	120	220	8	160	220	1	200	220	0	240	220	3	190	380
7	120	230	7	160	230	7	200	230	7	240	230	2	190	390

Table 4.8: RTHM component locations (X, Y) and types (CT) for verification example [48].

CT	**X**	**Y**	**CT**	**X**	**Y**	**CT**	**X**	**Y**	**CT**	**X**	**Y**
6	100	60	9	140	100	8	180	220	5	220	220
3	100	90	4	140	140	8	200	60	10	240	40
2	100	130	10	140	180	9	200	100	9	240	60
10	100	180	7	140	220	9	200	130	4	240	80
4	100	230	5	160	60	3	200	140	8	240	100
4	120	50	2	160	100	7	200	170	1	240	120
10	120	90	5	160	140	10	200	180	10	240	140
6	120	130	4	160	180	4	200	220	7	240	180
9	120	150	9	160	220	9	220	40	6	240	200
5	120	190	9	180	60	9	220	60	7	240	210
9	120	230	5	180	100	10	220	100	2	240	220
9	140	40	8	180	140	9	220	160			
2	140	80	4	180	180	7	220	200			

The RTHM problem, like the PAPM problem, requires both the solution of the component placement sequence as well as the component allocation to the feeder slots. The component locations and types are presented in Table 4.8 with the coordinates relative to the panel's lower, left-hand corner.

Note that in the AIM and PAPM cases the sole parameter of interest is distance traveled by the placement head, since any constant velocity would have no impact on the solution results for these two machine types. In the RTHM case, however, the three competing subsystems described in section 3.3.3 operate at different velocities. The distances traveled by these subsystems produce different time requirements for the placement of components in the placement sequence. A distance value is no longer applicable nor sufficient to describe the placement and allocation solution. Thus, the time required to populate the board is used as the solution and is given in seconds.

4.5 Experiment 2

This experiment was designed to examine the effect of off-center, or eccentric, component groups on PCB within a panel. The PCB layouts for these experiments are shown in Figures 4.5, 4.6 and 4.7. These figures display the different panel layouts—i.e., the number of PCB per panel—for which these three PCB design cases were applied. These component layouts were designed such that case A has the component population centered on the PCB geometric center. The second case B has the component population located half the distance from PCB geometric center and its boundaries. Case C has the component population located such that the outermost components of its distribution are at the outer corner of the PCB boundary. It should be noted that each time a PCB population changes (i.e., two-PCB verses four-PCB for a panel), the component locations relative to the PCB geometric center will be different for cases B and C, since the PCB dimensions decrease as the PCB population in the panel increases.

The components are equally spaced eight millimeters between centers. There are four component types per PCB; each type has four components in each PCB, for a PCB component total of 16. The PCB designs which have the offset, tightly grouped components are referred to as *eccentric* designs or layouts in this research for sake of brevity.

All three cases are examined as a set for each machine type/panel layout scenario. The results are reported as "per component" since the number of components per panel change for each PCB per panel scenario.

4.6 Experiment 3

This experiment was developed such that there is no eccentricity of the component locations, yet an eccentricity associated with the component types layout is apparent. The PCB component location and type layouts are illustrated in Figure 4.8. The component types are

labeled adjacent to each component location in Figure 4.8 for clarity.

This experiment is applicable only to the PAPM and RTHM machine types, since these machines account for different component types on the same PCB. Though the general component distribution is uniform across the width and length of each PCB, the component types are not. The components are equally spaced every 20 millimeters across the horizontal PCB axis and every 40 millimeters across the vertical axis. There are four component types per PCB; however, there are an unequal number of components per each type, thus unbalancing the component type distribution.

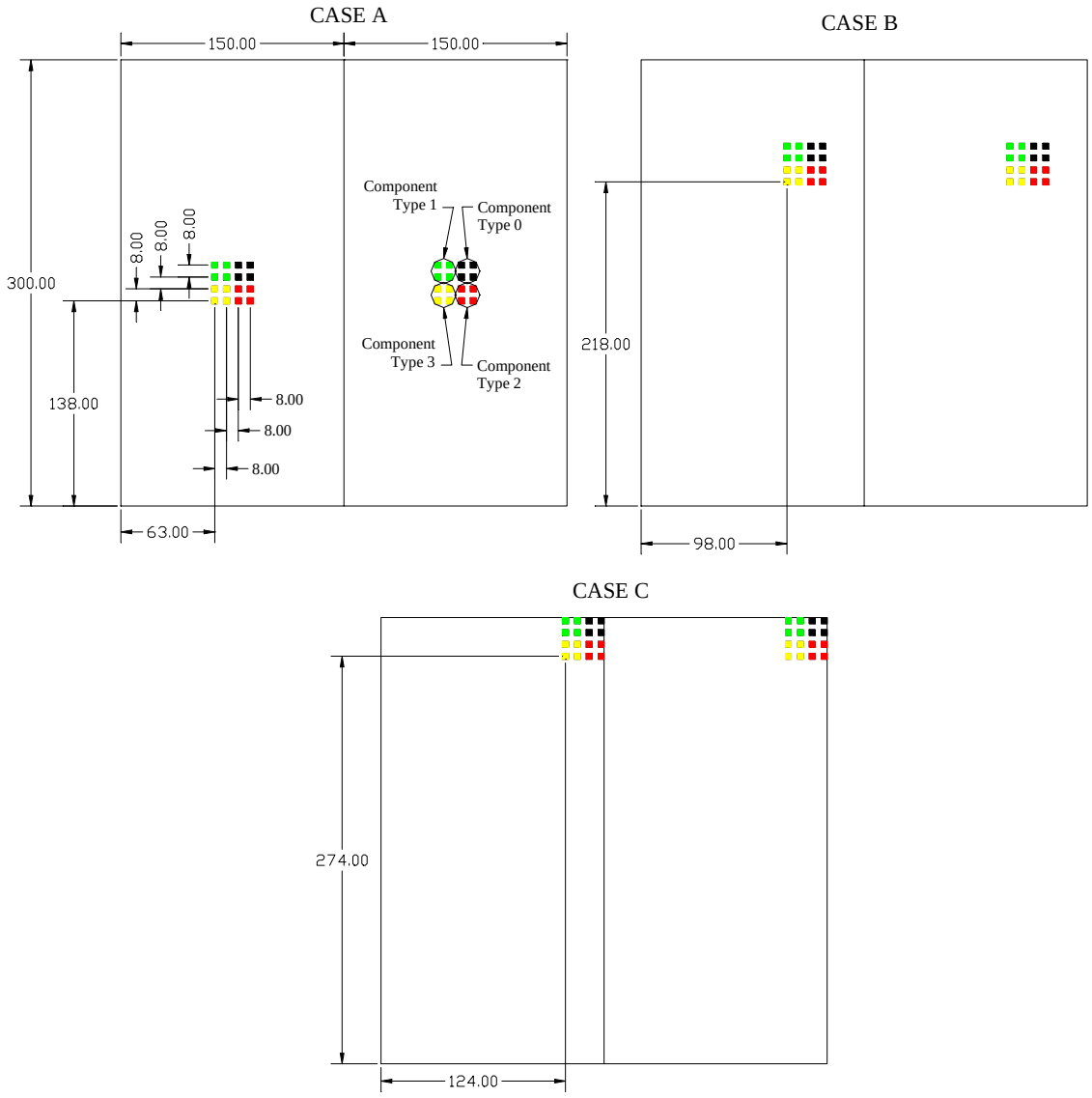


Figure 4.5: Experiment 2, Cases A, B and C for 2-PCB panels.

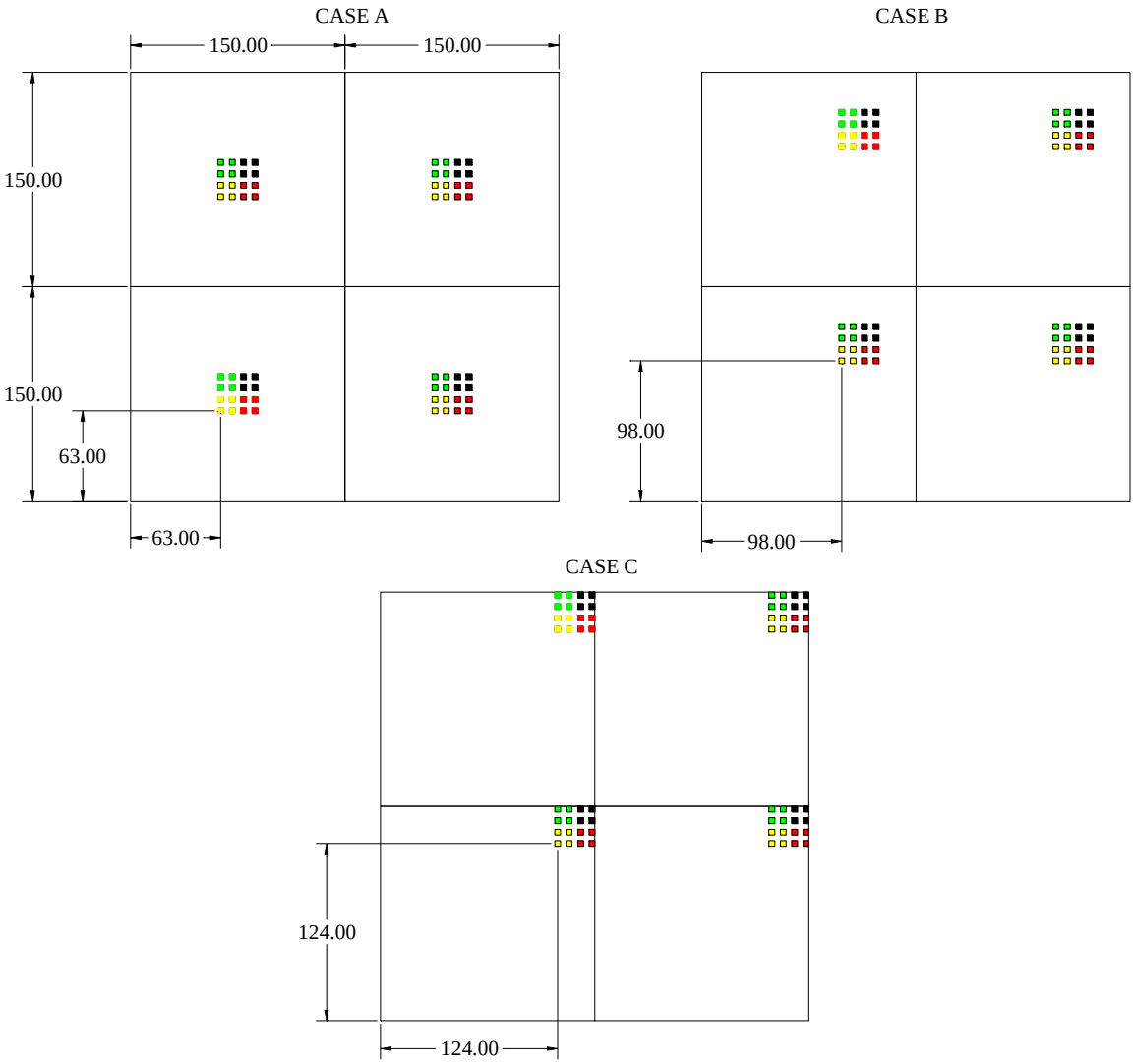


Figure 4.6: Experiment 2, Cases A, B and C for 4-PCB panels.

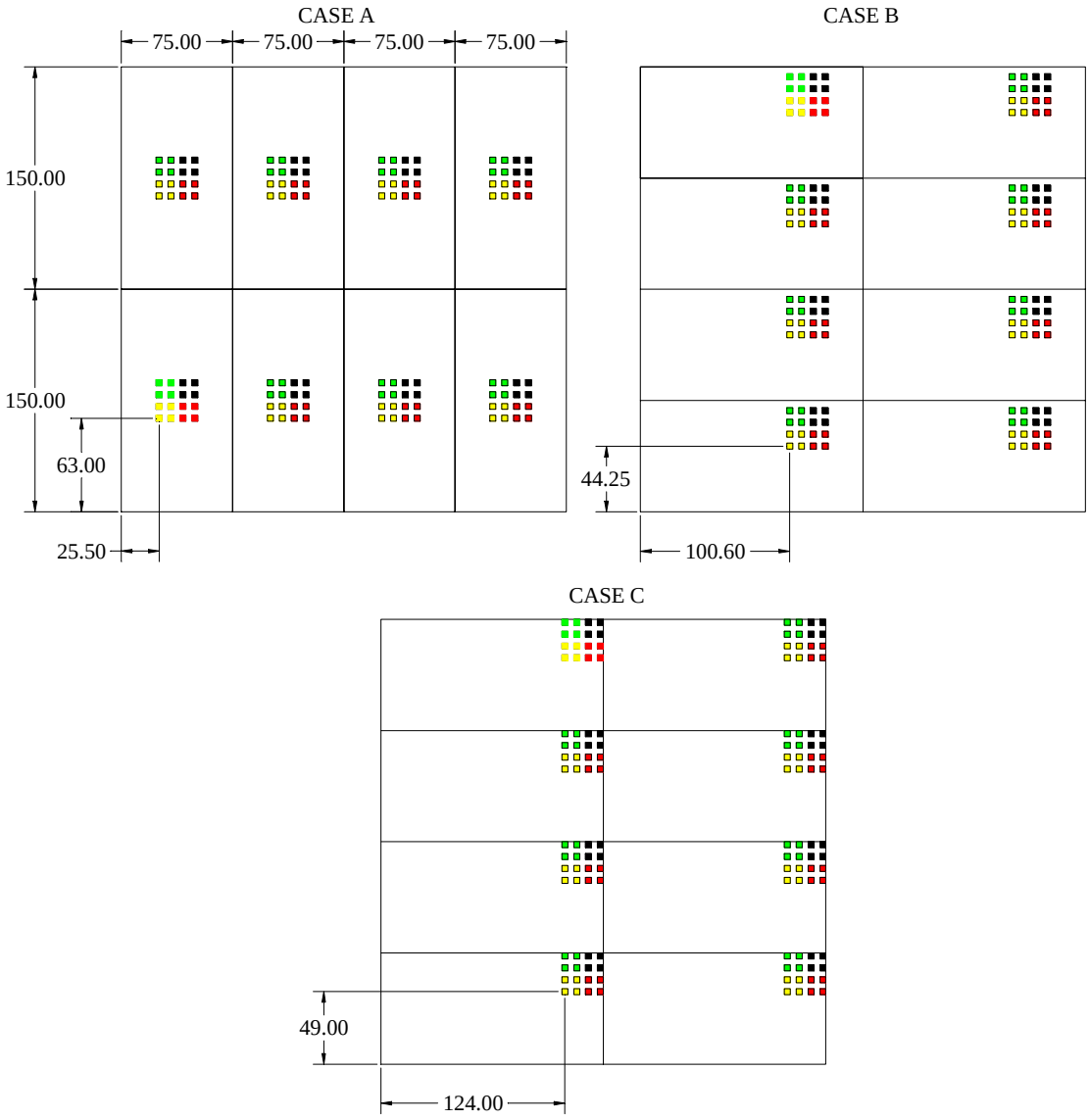


Figure 4.7: Experiment 2, Cases A, B and C for 8-PCB panels.

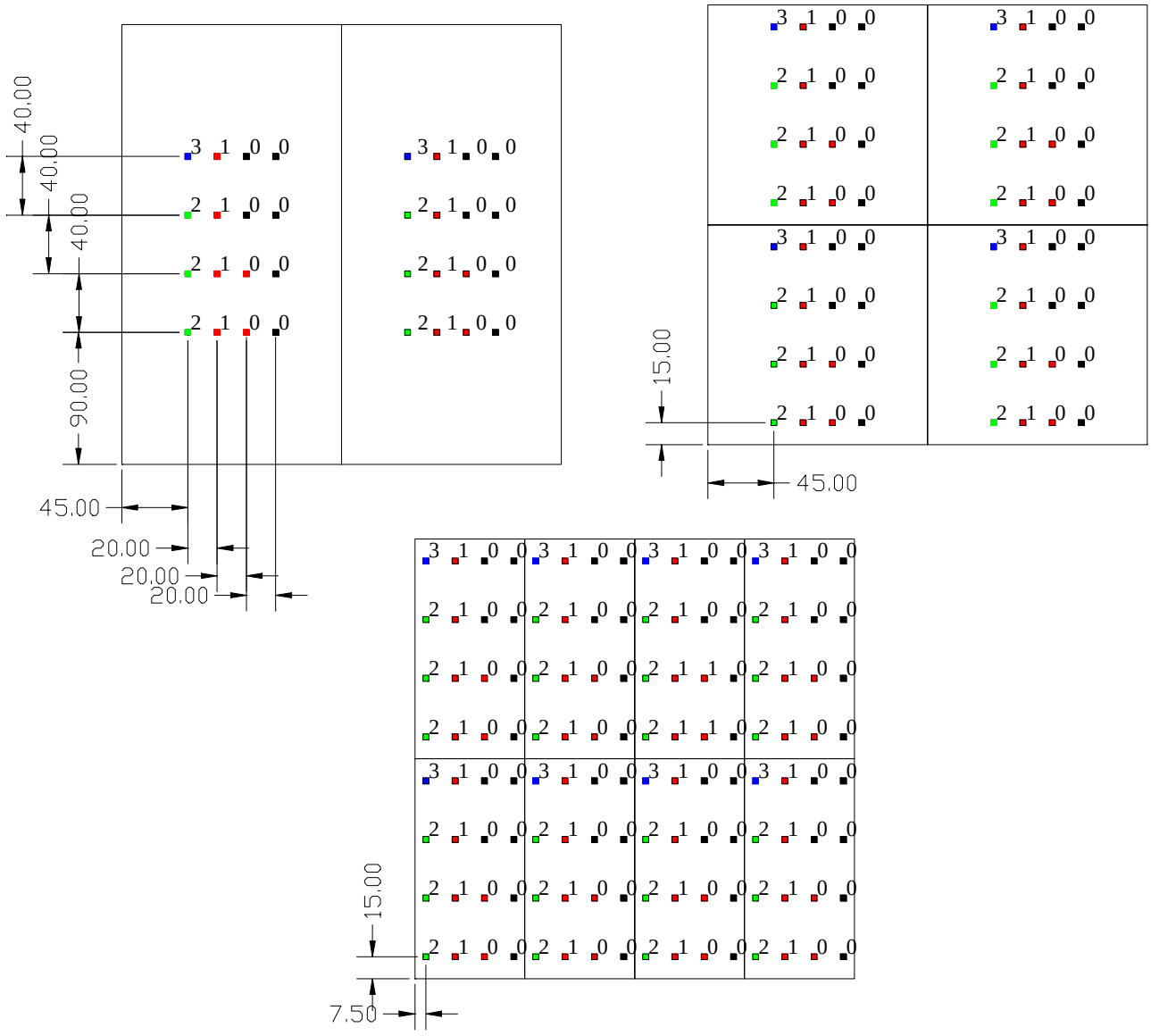


Figure 4.8: Experiment 3 for 2, 4 and 8-PCB panels.

4.7 Experiment 4

This experiment uses an example of a layout of a small PCB used in the mobile communications industry. There are 27 components for each PCB, with 20 component types, which are illustrated in Figure 4.9. There are zones clear of components on the PCB. These areas are where large components are placed by hand after the machine placements are accomplished. Since component types are present, only the PAPM and RTHM are considered for placement machines. This device is currently manufactured in a two-row by four-column layout as sketched in Figure 4.10. The PCB dimensions are such that the 36 patterns possible for the PAPM situation are the same as those illustrated in Appendix F and also match the 12 pattern alternatives for the RTHM (Figure 4.4).

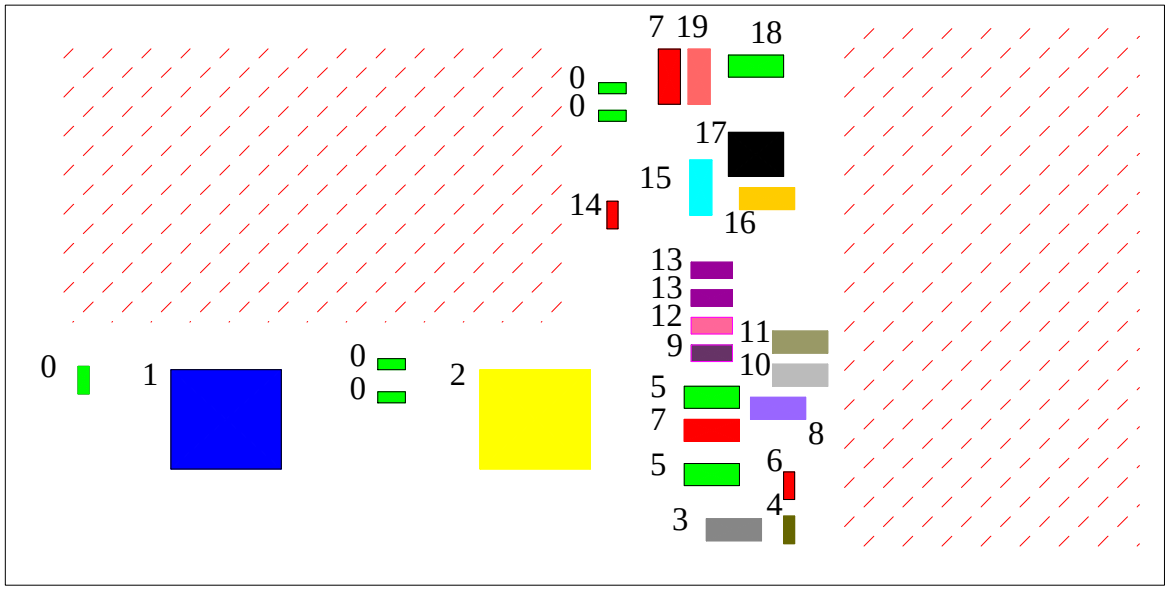


Figure 4.9: Experiment 4, component types

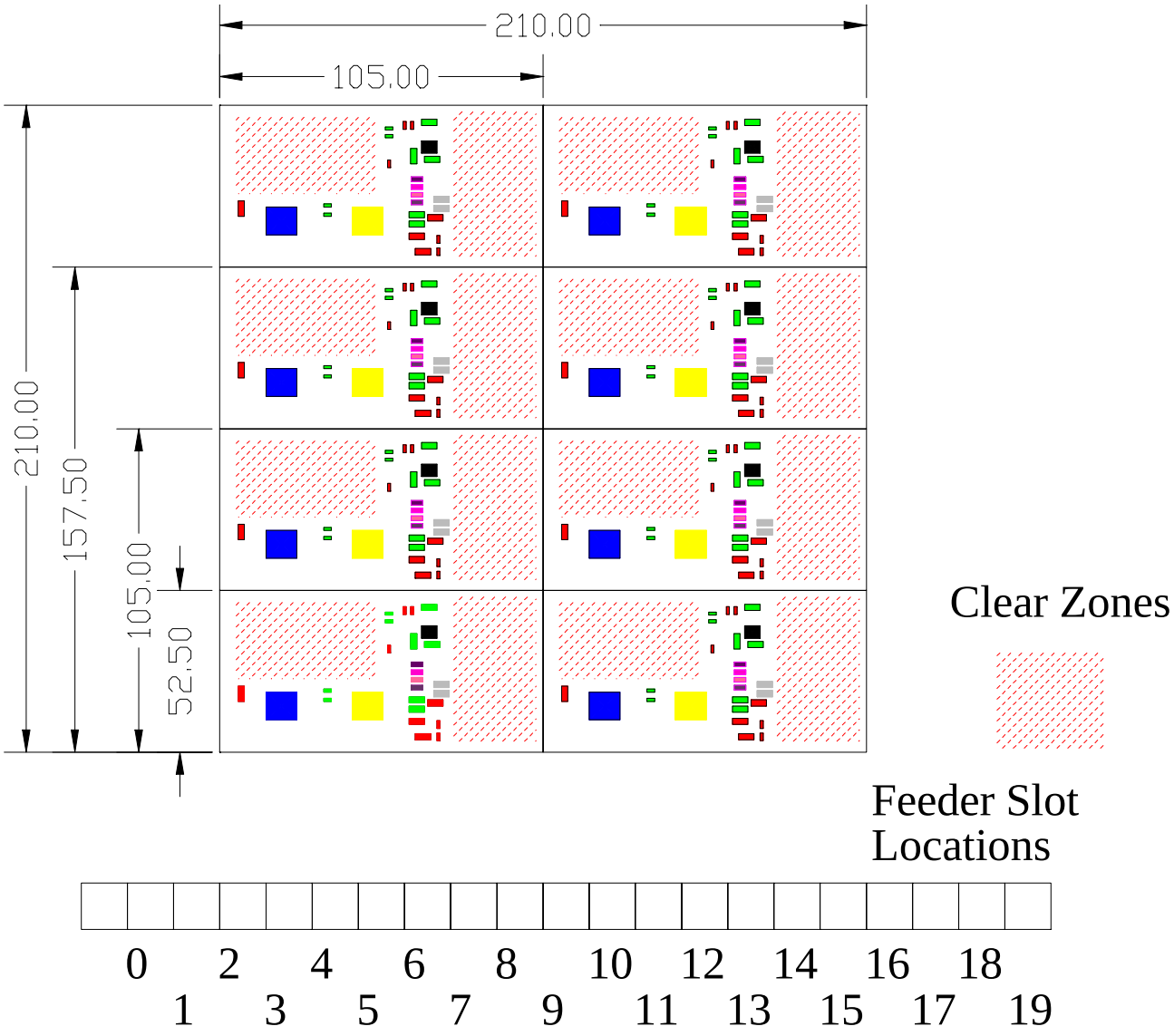


Figure 4.10: An industrial example of a panelized PCB layout (Experiment 4).

4.8 Experiment 5

The last experimental case may be considered the most general scenario under which panelization is examined. The component layout is generated randomly within the PCB boundary. The number of components and the number of component types are randomly selected from a range of values, based upon the survey of component placement literature in section 4.1. The following factors were considered of interest:

1. Low of 100 to high of 150 components per panel.
2. Low of 5 to high of 10 component types per panel.

This random-PCB design experiment is conducted somewhat differently than the previous experiments, as component locations are randomly generated for the PCB by `panelizer` via command-line options (see Appendix A). The procedure for generating comparisons is as follows:

1. Patterns are generated for the case at hand via `pmaker` (Appendix B).
2. Ten different PCB component layouts are generated in the process of conducting the global analysis. The global GA analysis is repeated three times with different random number seeds for each repetition. The best global results of three repetitions for each layout are recorded.
3. The resulting component layouts are written to files for use in the traditional analysis to come.
4. The "worst" and "best" panel designs for each layout are determined via `finder` (Appendix C), using the component layout files from step 3.
5. The Estimated Worst and Estimated Best panel designs are run in a traditional GA analysis to find best component placement times for those experimental instances. These analyses produce the Traditional Best and Traditional Worst results.

6. The traditional results are paired with the corresponding global analysis results for the same experimental instances. The values are statistically analyzed under two hypotheses: $H2_o : \hat{TW} - \hat{TB} > 0$ and $H1_o : \hat{TW} - \hat{G} > 0$.

where $\hat{TB}$, $\hat{TW}$, and $\hat{G}$ are the average values of the Traditional Best, Traditional Worst and global results, respectively, for the ten different random PCB layouts. $H1$ represents the comparison of the two traditional results (Best and Worst) and $H2$ the comparison between the Traditional Worst value to the global results. The Traditional Worst value is used for comparison against the global analysis results because there is no other value against which to compare. $H1_o$ and $H2_o$ correspond to the TLPI and GLPI described in section 4.3, respectively, but from a statistical viewpoint. It is anticipated these comparisons should be equivalent for the same, randomly-generated, PCB designs, though the best results were generated by two different techniques.

Since several computer runs are required for each experimental sample, only ten experimental instances are conducted for each machine type and PCB per panel situation. A moderately large confidence interval must be used as a result, in order to yield any useful results for the large number of situations (number of PCB and all machine types) for the study at hand. A confidence interval of 90% was used, based upon the above considerations.

The purpose behind this experiment is to statistically determine if the most general case of a PCB design (i.e., randomly placed components and types) can show a difference in panel design alternatives (via the LPI results) through the implementation of panelization to the component placement problem. A positive confidence interval would represent clear evidence that panelization will present some potential for improvement of a good panel design selection over that of a bad selection for that experimental situation.

Each of the experimental samples have the Estimated Best, Estimated Worst, Traditional Best, Traditional Worst, and global computer runs. The Estimated Best and Worst results are obtained from a single run of `finder`, but the Traditional Best, Traditional Worst and Global each require a separate run of `panelizer` in order to obtain solutions for a par-

ticular experimental instance. There are 90 experimental instances as illustrated in Figure 4.5, yielding a total of 360 GA computer runs which could produce graphical output. Due to such a large number of illustrations, the documentation of such graphics is not given as part of Chapter 5.6. The numerical results (ELPI, TLPI and GLPI) are be reported appropriately in tables for comparisons. A statistical analysis table follows the results tables in order to report the confidence intervals describing $H1_0$ and $H2_0$.

Chapter 5

Experimental Results and Discussion

This chapter presents the experimental results and discussion of these results. Due to the large amount of graphical data presented with each experiment, a brief discussion for each experiment is presented with the tabular and graphical results presentation. A discussion of the results as viewed across experiments 2 through 5 is presented section 5.6.

5.1 Experiment 1 Results

A summary of experiment 1 results for all the machine types is given in Table 5.1.

5.1.1 AIM

The baseline against which the experiment is validated is presented as the "optimal" solution by Leu et al.; the score was 256.8 millimeters. The placement progression is displayed in Figure 5.1. The `panelizer` program was used against the same problem in a traditional

Table 5.1: Experiment 1 results.

MACH	*Baseline [48]*	*Research Results*
AIM	256.8 mm	255.8 mm
PAPM	61,900 mm	61,852 mm
RTHM	52 sec	26 sec

approach and yielded a best score of 255.8 millimeters; the path generated is illustrated in Figure 5.2. The results show that the `panelizer` program converges to the lower values produced by the published literature. Its disadvantage is that the convergence took longer to obtain that the published problem (420 generations verses 200 generations). The convergence of `panelizer` for the AIM case is compared with that of the published case by Leu et al. in Figure 5.3. Genetic algorithms have as their basis an inherent randomness associated with their population initialization, parent selection, and crossover operation; slightly different techniques used for the implementation of these techniques will result in different convergence results for identical problem solutions [17]. The published example in Leu et. al. gave only a general overview about how their particular GA program was constructed. Thus, it is likely that unknown differences in the baseline example and `panelizer` led to the different convergence rates for the same solution.

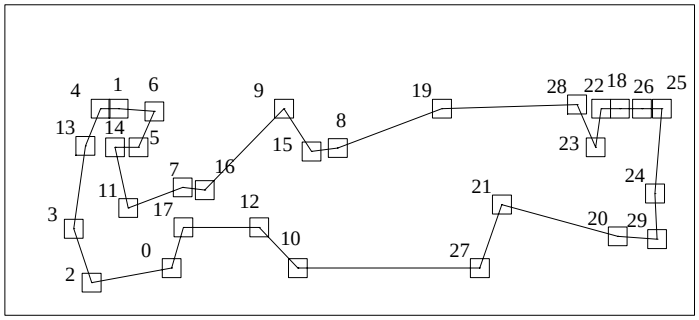


Figure 5.1: Best solution sequence for AIM problem from Leu et. al. [48].

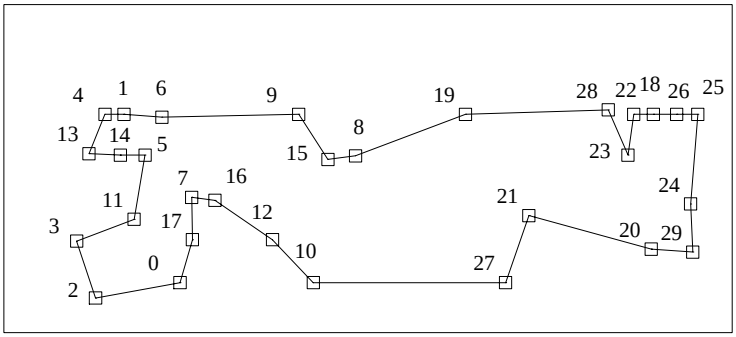


Figure 5.2: Best solution for the AIM experiment 1 problem using `panelizer` .

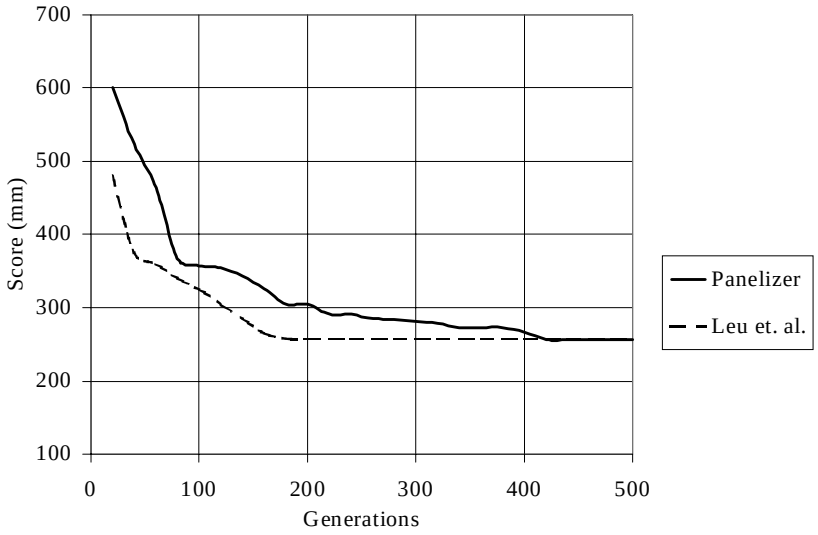


Figure 5.3: Comparison of `panelizer` convergence against Leu et. al. [48] for AIM experiment 1.

5.1.2 PAPM

The "optimal" result given by Leu et al. was approximately 61,900 millimeters. The `panelizer` program was used against the same problem and yielded a best score of 61,852 millimeters; the path generated is illustrated in Figure 5.4, where the arm motion paths were separated into the required and unrequired motions to reduce clutter of the illustration. As for the AIM verification experiment, `panelizer` was set up for the traditional approach such that the panel had only one pattern available and all the component locations were not allowed to move. The convergence of `panelizer` for the PAPM case is compared with that of the published case by Leu et al. in Figure 5.5. The convergence was much faster for `panelizer` than that shown in the published literature, presumably for the same reasons discussed in section 5.1.1.

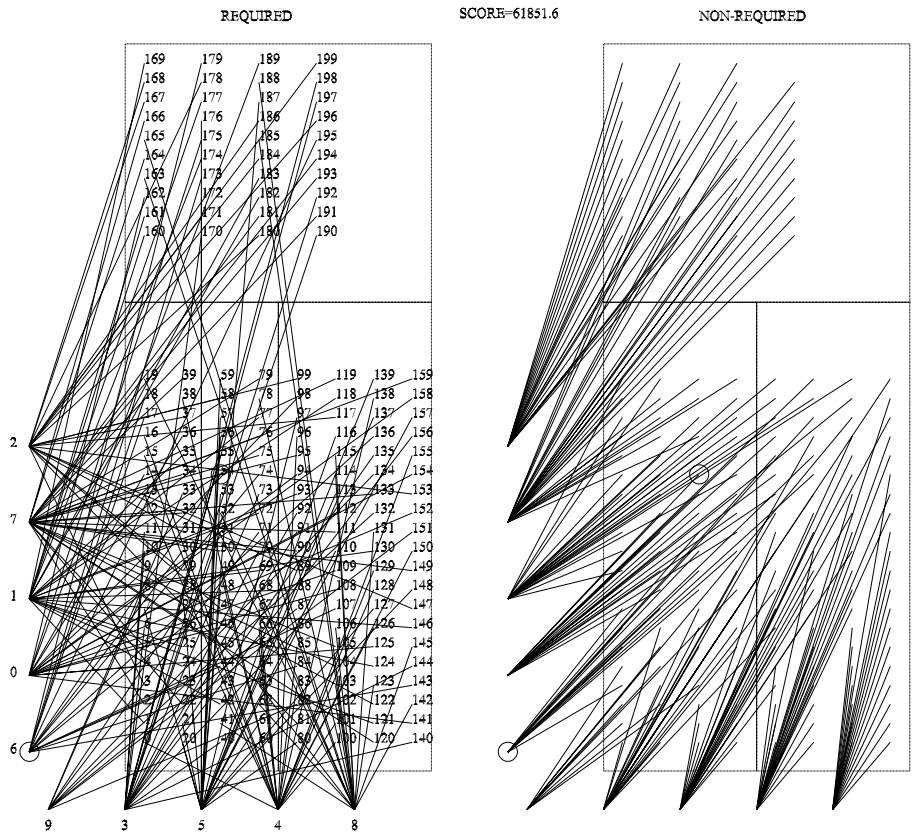


Figure 5.4: Best solution sequence for PAPM experiment 1 from `panelizer`.

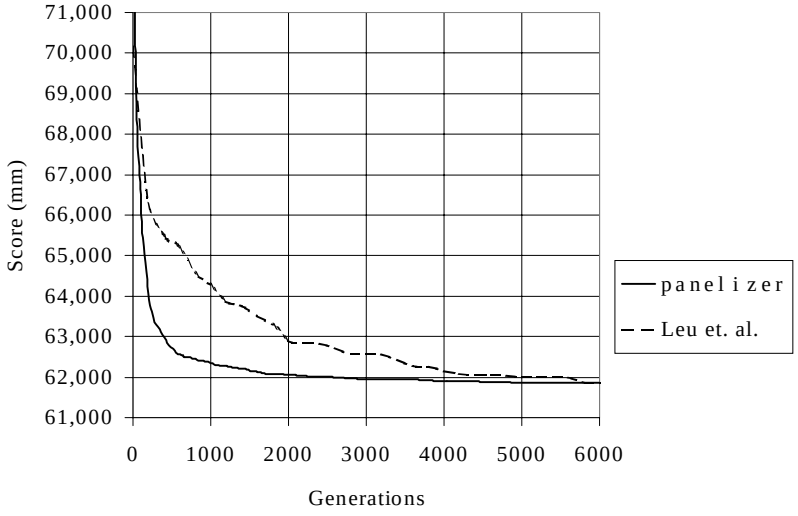


Figure 5.5: Comparison of `panelizer` convergence against Leu et. al. [48] for PAPM experiment 1.

5.1.3 RTHM

The placement time solution given by Leu et al. for a specific case was approximately 52 seconds after 4000 generations; this value was not the value reported in the citation text [48], but there were several errors in that publication, relative to this particular machine case. For details on that error and corrections, refer to Appendix D. The `panelizer` program was used against the same problem and yielded a best score of 26 seconds for the same number of generations; the table travel path and the feeder allocation is illustrated in Figure 5.6. Though the feeder slots are shown as being a distance below the panel, in reality, their location relative to the panel is not necessary for the problem structure.

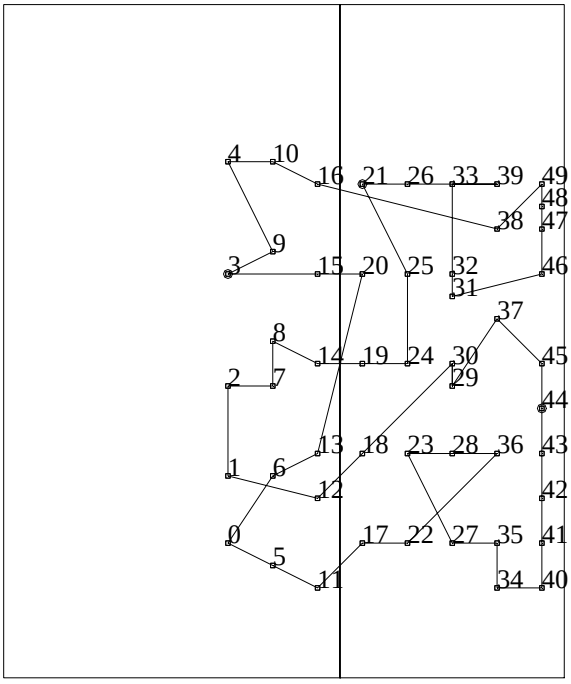


Figure 5.6: Best solution sequence for RTHM experiment 1 from `panelizer`.

The convergence of `panelizer` for the RTHM case cannot be compared with that of the published case by Leu et al., since the published convergence information was in error (Appendix D). Therefore, only the solution trends of the `panelizer` results are given in Figure 5.7 through 4000 generations. The solution trend appears to match that of the other cases, in that a gradual decrease from the initial population solutions is apparent. It is also apparent that the entire 4000 generations are not necessary to reach a converged value for the problem.

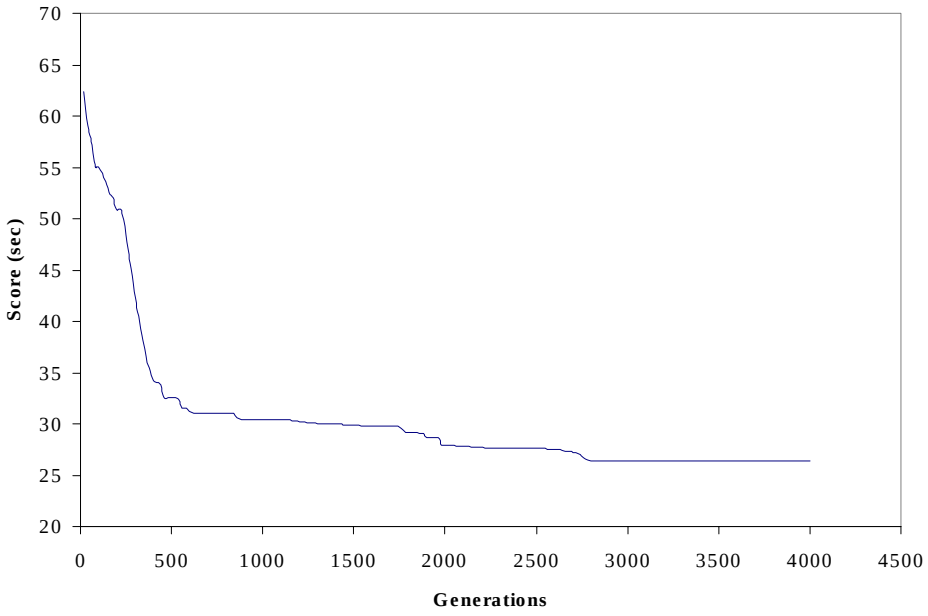


Figure 5.7: Solution trend for RTHM experiment 1 case.

5.2 Experiment 2 Results

A summary of all the data from experiment 2 is given in Table 5.2. The raw data and the panel designs resulting from these analyses are given in Appendix E. The graphical illustrations of the solutions are presented collectively for each machine type at the end of this section.

Table 5.2: Experiment 2 summary of results.

MACH	*No. of PCB*	*CASE*	*Estimated Worst*	*Estimated Best*	*Traditional Worst*	*Traditional Best*	*Global*	*ELPI*	*TLPI*	*GLPI*
AIM	2	A	150.00	150.00	15.58	15.58	15.58	0.00%	0.00%	0.00%
		B	270.40	150.00	22.81	15.61	15.99	44.53%	31.57%	29.90%
		C	387.50	150.00	30.16	15.60	15.78	61.29%	48.28%	47.68%
	4	A	424.26	424.26	16.89	16.89	16.97	0.00%	0.00%	-0.48%
		B	636.40	212.13	21.28	11.62	11.96	66.67%	45.40%	43.79%
		C	774.99	73.54	23.96	7.09	7.91	90.51%	70.41%	66.98%
	8	A	876.24	840.83	17.95	16.45	17.45	4.04%	8.36%	2.83%
		B	1194.58	571.59	19.71	16.23	15.24	52.15%	17.65%	22.67%
		C	1376.49	429.36	20.70	13.60	12.91	68.81%	34.32%	37.64%
PAPM	2	A	177.25	160.39	336.06	305.26	313.03	9.51%	9.17%	6.85%
		B	249.50	117.25	477.93	216.60	221.23	53.01%	54.68%	53.71%
		C	312.41	83.20	600.29	148.22	152.62	73.37%	75.31%	74.58%
	4	A	180.77	180.19	345.65	344.98	349.65	0.32%	0.20%	-1.16%
		B	230.63	131.10	443.65	250.01	253.17	43.15%	43.65%	42.93%
		C	264.11	100.93	509.31	193.64	196.56	61.79%	61.98%	61.41%
	8	A	183.95	180.28	355.62	350.08	352.13	1.99%	1.56%	0.98%
		B	221.06	146.58	430.08	283.34	285.37	33.69%	34.12%	33.65%
		C	244.32	127.54	476.54	245.23	246.62	47.80%	48.54%	48.25%
RTHM	2	A	75.43	75.43	0.200	0.200	0.200	0.00%	0.00%	0.00%
		B	135.45	75.43	0.254	0.200	0.200	44.31%	21.32%	21.32%
		C	193.91	75.43	0.309	0.200	0.200	61.10%	35.34%	35.21%
	4	A	106.37	106.37	0.202	0.201	0.209	0.00%	0.40%	-3.72%
		B	159.30	53.63	0.238	0.173	0.165	66.33%	27.37%	30.65%
		C	193.91	19.99	0.260	0.150	0.150	89.69%	42.40%	42.28%
	8	A	109.84	105.51	0.221	0.199	0.195	3.93%	9.79%	11.60%
		B	149.54	71.99	0.232	0.198	0.207	51.86%	14.47%	10.50%
		C	172.25	54.65	0.235	0.175	0.189	68.27%	25.39%	19.65%

5.2.1 AIM

The AIM machine type presents a TSP problem, discussed in section 3.3.1. There are less patterns for an AIM problem than for the PAPM machine, due to the neglecting of bias associated with the machine fixture orientation, relative to the panel. This restriction was first discussed in Section 3.1.

Experiment 2, 2-PCB

There is only one possible pattern for the AIM problem with two identical PCB, but four PCB rotation sets are possible. The Estimated Best and Worst solutions are shown in Figure 5.9. This figure shows that the best and the worst PCB rotation sets for case A are identical. Therefore, it is not surprising that the panelization approach to the problems will show little to no improvement over that of the traditional approach. Figure 5.9 shows that the global approach panel design solution differed in the PCB rotation set from that of the traditional best approach. This situation highlights the fact that a PCB with complete symmetry about its geometric center will likely show little panel design preference, relative to a component placement solution.

The identical panel designs for all experiments for case A produced the same final score of 15.58 mm/component. However, there is approximately a 2% difference in the TLPI verses the GLPI for the identical panel designs of cases B and C. Section 4.3 formulated the TLPI and the GLPI as two measures for the largest potential for improvement through two different analysis techniques. Thus, these values should be the same for the same experimental instance. This discrepancy is explained by the fact that these highly eccentric cases result in long moves which must be made as the head travels from one group of components (on one PCB) to the next. The GA has the tendency to settle upon a set of longest moves which may not be the most optimal set for the panel design at hand. This characteristic is illustrated in Figure 5.9 for case A and B in the global and traditional best situations.

As the eccentricity of the PCB layout decreases, one can expect this type of discrepancy associated with the longest moves to also decrease. Based upon the observations thus far, for experiments involving cases B and C one must consider the LPI to be significant only if greater than 2%.

It is of importance to note that the *best* raw score for a panel with two PCB will not necessarily increase due to the increase in the eccentricity of the PCB component population centers. This fact is most clearly illustrated in Figure 5.9, where the Traditional Best and global scores are within 2% of each other, yet the PCB component population centers are in different locations for the different cases. This result was anticipated by the estimator function results give in Figure 5.8.

The ELPI was developed as a potential measure for anticipating the value of the panelization approach, even before the panelization analysis was applied (section 4.3). In this scenario, the ELPI appears to loosely predict the TLPI or GLPI, though the margin of error high for case B (roughly 13% higher than the 31% TLPI score, for example). However, the same margin is present for the case C situation, indicating a potential for correlation between the ELPI and the two LPI measures.

Even though the global and Traditional Best scores do not increase from case B to C, the LPI is greatly increased as one progresses from case A to case C, since the Traditional Worst scores become progressively larger. This simplest panelization scenario of two, rectangular PCB in a panel results in a single pattern and only four PCB rotation sets; yet, there are LPI of up to 48% in the worst-case scenario, case C.

In simplest terms, the choice of a bad panel design can have progressively worse impact upon the solution if a traditional minimization approach is used without considering the panelization aspects of the problem. This appears especially true if eccentric PCB layouts are involved.

Experiment 2, 4-PCB

The estimator results are illustrated in Figure 5.10. The component placement sequences for both the global and traditional analyses are illustrated in Figure 5.11.

Observations for the four-PCB panel scenario are similar to that of the two PCB scenario. As for the two PCB panel, there is only one possible pattern for the AIM problem with four identical PCB, but there are more rotation sets (16) since the PCB are square. The GLPI shows a very small negative result; it is not considered unusual, since it has already be observed that the panelization approach does not necessarily produce potential improvements for the case A scenario.

The case A Traditional Best, Traditional Worst, and global scores are nearly identical; this situation is a repeat of the two PCB situation and is expected, since the PCB show symmetry in their PCB layouts for the four-PCB panel of case A. However, since there are several long moves possible between the PCB component groups, the case A Traditional Best and global scores vary by 0.5% with respect to each other.

The panel experiments explored thus far with four PCB differs from the two PCB panel designs in that the best value discovered for cases B and C increases as the PCB component layout increases its eccentricity. This increase is a direct result of the panelization geometry. As seen in Figure 5.11, the clusters of component populations have a greater opportunity for gathering closer together as the eccentricity increases. The estimator functions were developed to anticipate this opportunity for improving the LPI. This situation also accounts for the larger LPI than found in the two-PCB scenario; the four-PCB scenario yields a maximum of 67% GLPI and 70% TLPI in case C.

The ELPI are increasing as the two LPI measures increase. Similar to the two PCB scenario, the ELPI has a fairly constant "error" for cases B and C. This error is about 20% in the four-PCB situation for these two cases.

Experiment 2, 8-PCB

The Estimated Best and Worst results are illustrated in Figure 5.12. The component placement sequences for the global and traditional analyses are illustrated in Figure 5.13.

The eight-PCB situation allowed for 12 patterns as part of the panel designs; these patterns were previously illustrated in Figure 4.4. This added complexity to the problem allows for differences in the estimator values between some patterns even for case A, as shown in Figure 5.12. There is an apparent estimator difference of 4% between the Estimated Best panel design and the Estimated Worst for case A. There is also a 4% or less LPI when the scores of the global or Traditional Best analyses are compared against the Traditional Worst scores. Though the score differences are similar to the estimator differences, some portion of this difference could be attributed to the usual variance associated with the GA algorithm settling at similar lower values (section 5.2.1), since the GA has been shown earlier to settle at different (though similar) values for identical panel designs with PCB component layouts that are clustered about the PCB geometric center.

The LPI for each case increased when comparing the two PCB scenario to the four-PCB scenario. However, when viewing the eight-PCB scenario in this comparison, it is discovered that the LPI increases did not continue. Both the TLPI and the GLPI did increase as one increased the PCB layout eccentricity, however. The maximum TLPI and GLPI was 34% and 38%, respectively, for case C. The ELPI values corresponded with the LPI measures for each case as well, with the discrepancy between this predictor and the LPI being roughly 25%.

5.2.2 PAPM

The PAPM situation is described in section 3.3.2. There are more patterns for the PAPM situation than for the AIM, due to the bias associated with the machine fixture orientation, relative to the panel and the feeder slots located in the machine. Thus, there are 2 patterns

available for the panel design alternatives.

Experiment 2, 2-PCB

The Estimated Best and Worst results are presented graphically in Figure 5.14. An illustration of the global and traditional solutions is shown in Figure 5.15. Only the "nonrequired" moves are drawn in order to reduce clutter; this shorthand method of illustration looses no accuracy in representation, since "required" moves are the same for any given panel design.

Though all the head moves may be considered as relatively long with respect to the distances between the components, variations between these moves produce very little impact on the variability of the final score for identical panel designs. This observation is made with respect to those made for the AIM case, where the variable long moves could cause up to a 2% difference in solutions for identical designs. This lack of impact from the differences in the PAPM long moves is a characteristic of these experiments particular to the machine type, since all the feeder slots are close together (relative to the component spacings within each PCB) and only on one side of the panel. If multiple feeders were present on more than one side of the panel, then this situation could change. However, that aspect of experimentation was not investigated in this research.

The two PCB panel for a PAPM allows for two patterns. Even for case A, a particular pattern was preferred by the estimator prediction and the global approach. The global and Traditional Best scores differed by only about 2% for each case A, B and C. The TLPI and GLPI stayed within 2% of each other for all the cases, and increased as the eccentricity of the PCB layout increased, to a high of 75% for both the TLPI and the GLPI for case C. The ELPI was also very close to the TLPI and GLPI figures for all cases.

Experiment 2, 4-PCB

The Estimated Best and Worst results are presented graphically in Figure 5.16. An illustration of the global and traditional solutions is shown in Figure 5.17. Only the nonrequired moves are shown to reduce visual clutter.

The four-PCB panel for a PAPM allows only one pattern. Thus, the PCB rotations were the only difference in panel design alternatives. Even so, a very slight difference in the Estimated Best and Worst results is discovered. This result is entirely due to the location of the component types on the PCB, though the difference is very small. For case A, the TLPI and LPI both are below 2%. However, as the PCB layout eccentricity increased, both the global and Traditional Best scores decreased while the Traditional Worst score increased. The highest TLPI and GLPI was registered for case C at 62% and 61%, respectively. The ELPI again closely predicted the two LPI values to within 1%.

Experiment 2, 8-PCB

The Estimated Best and Worst results are illustrated in Figure 5.18. The component placement sequences for both the global and traditional analyses are illustrated in Figure 5.19. In this panel scenario, there are 36 patterns available for the panel design alternatives; these patterns are illustrated in Appendix F.

The eight-PCB PAPM cases, as with the AIM eight-PCB cases, have both multiple patterns and PCB rotation sets as part of the panelization problem. The Estimated Best and Worst results for case A differ, though that difference is less than 2%. Since earlier GA examples suggest that differences of less than 2% are not distinguishable as a true differences in the panel design, these results would ordinarily not be assumed to be significant. The TLPI and GLPI match the small, difference predicted by the ELPI. There was an increase in the GA results for both analysis techniques as the PCB component layout increased, to a maximum of 48% for both TLPI and GLPI for case C. The ELPI predicted the TLPI and

GLPI within 1% for each case.

For cases B and C, it should be noted that of all the 9216 panel design alternatives (36 patterns and 256 rotation sets) in each experiment, the global analysis panel design solution matched the estimator (and thus the best traditional) pattern and PCB rotation set. This is see by examining Figure 5.19. Matches of the two were also present in the two and four-PCB panel experiments.

5.2.3 RTHM

There are less patterns for a RTHM problem than for the PAPM machine, due to the lack of bias associated with the machine fixture orientation, relative to the panel. This situation is the same as for the AIM problems and can be related to the TSP portion of the overall RTHM problem structure.

Experiment 2, 2-PCB

The Estimated Best and Worst results are presented graphically in Figure 5.20. An illustration of the solutions is shown in Figure 5.21. It should be noted that though there is a TSP element to the RTHM problem, what may be the best TSP path in a solution may not be the best solution for the RTHM problem. This is a result of the three competing subsystems described in section 3.3.3, the impact of which are discussed at the end of section 4.3. Thus, the graphical illustrations from the GA solutions may not appear as organized as those from the AIM experiments.

The RTHM two PCB experiments appear similar to that of the AIM two PCB experiments, in that the global and Traditional Best scores follow each other closely, and the Traditional Worst scores are associated with the panel designs which have the PCB component clusters at opposite ends of the panel. The TLPI and GLPI both increase from 0% for case A to 35% for case C.

The ELPI for cases B and C increase along with the TLPI and GLPI, but have a considerable error with respect to the AIM experiments. In the RTHM two PCB situation, ELPI is higher than the TLPI and GLPI by 23% and 26% for case C.

Experiment 2, 4-PCB

The Estimated Best and Worst results are illustrated in Figure 5.22, and the solution sequences for both global and traditional approaches are shown in Figure 5.23.

The estimator scores for the best estimates get particularly small for case C (19.98), since the component clusters have the opportunity to group adjacent to each other. The percent difference between the best and worst estimator values is thus rather high, about 90%. This extreme difference in estimator values may be a result of the geometry of the experiments; the large difference was also becoming apparent for case C in the two PCB experiments as well.

The TLPI was 0% for case A, as may be expected for that case. However, there was a -3.5% GLPI for case A, even though the global, Traditional Worst and Best panel designs were basically the same. There were differences between the global and Traditional Best scores for both eccentric PCB layout cases; 4% difference for case B and 0.2% for case C. The 4% difference for case B is particularly noticeable, considering that the solution settled on two long paths that the global solution did not use. This lack of consistency between the two approaches nevertheless is well below the TLPI (27%) and GLPI (31%) in this situation. The ELPI increases with the PCB layout eccentricity; however, the ELPI increases its departure from the TLPI and GLPI percentages, being roughly 35% greater over cases B and C.

Experiment 2, 8-PCB

The Estimated Best and Estimated Worst results are illustrated in Figure 5.24, and the solution sequences are shown in Figure 5.25.

For case A, the eight-PCB situation allows for multiple patterns to contribute to a non-zero ELPI. The corresponding TLPI and GLPI have positive values which actually exceed the ELPI results. The TLPI and GLPI show increases as the cases progress, to a maximum of 25% and 20%, respectively, for case C.

Similar to the four-PCB experiments, there is some difference between the global and the Traditional Best scores for the same cases: -2.5%, 4.4% and 7.8% with respect to the global for cases A, B and C, respectively.

It appears that the RTHM experiments show a tendency to loose consistency for the best solutions as the number of PCB increase for the offset distribution PCB designs. This trend is counter to the AIM and PAPM results, where the global and Traditional Best results were within 2% for their respective three cases.

As in the two and four-PCB experiments, the ELPI is greatly outdistancing the TLPI and GLPI for cases B and C, though the ELPI does increase as the other two measures increase. The discrepancy between the ELPI and the TLPI is as high as 43% for case C. ELPI and GLPI have an even greater difference of 48% for the same case.

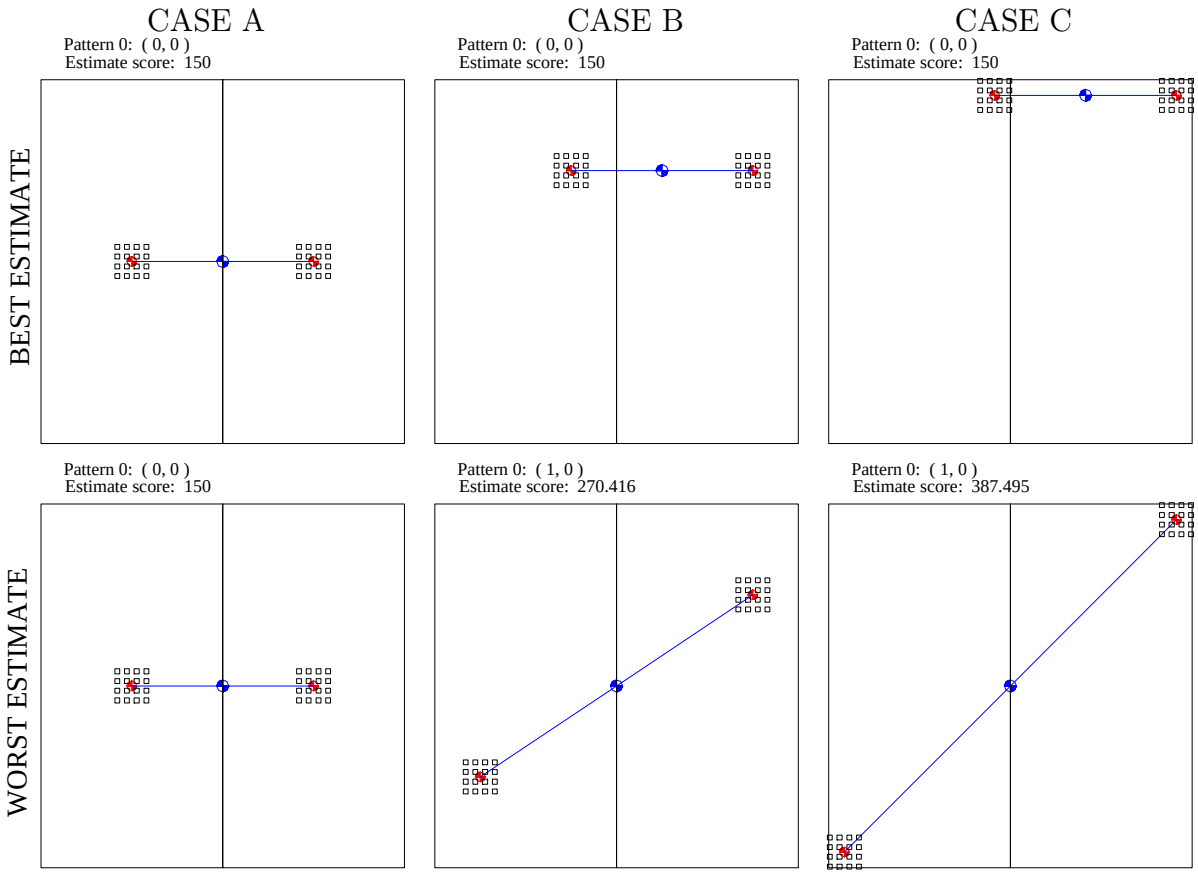


Figure 5.8: Best and Worst Estimator results for 2-PCB, AIM experiment 2. Values in mm/component

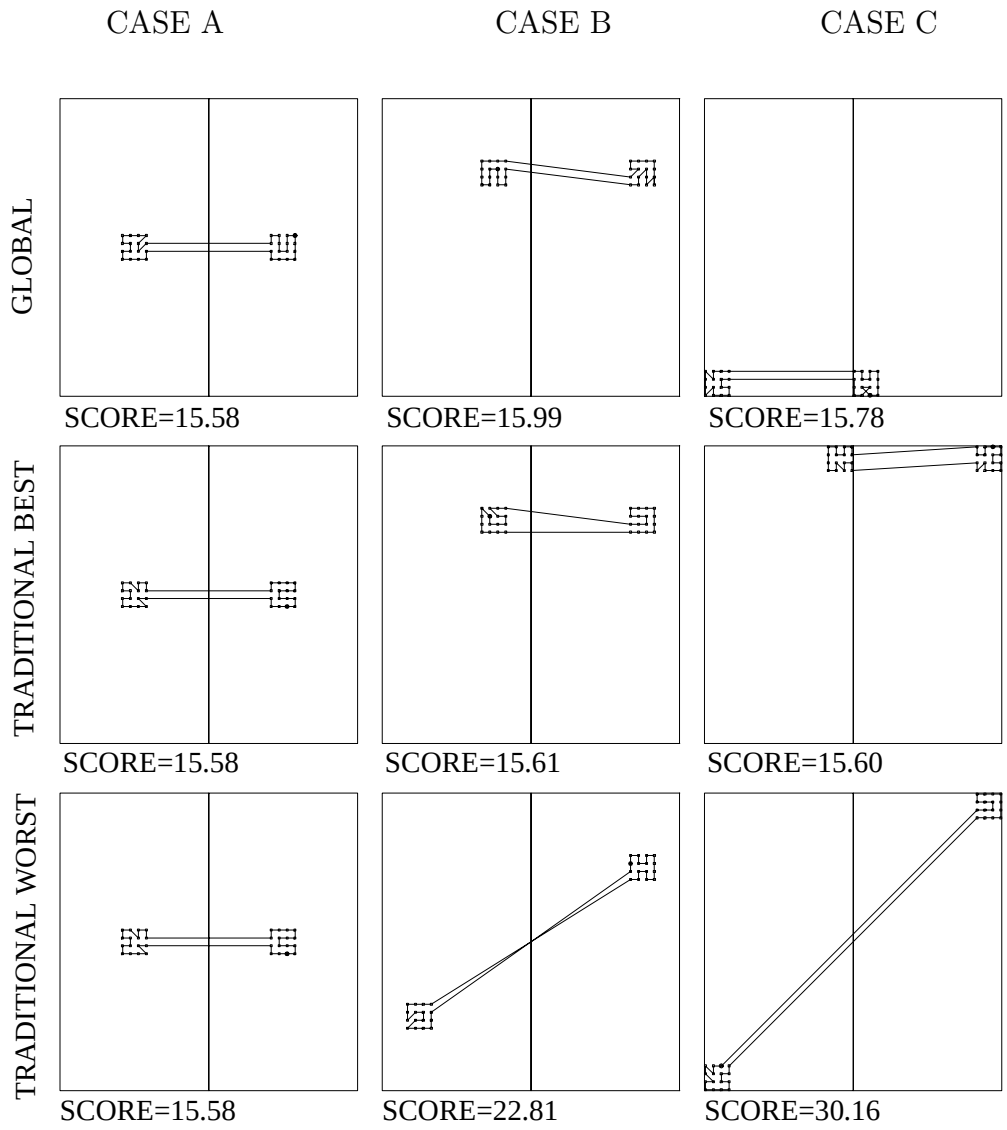


Figure 5.9: Solutions for AIM experiment 2 for 2-PCB. Scores in mm/component.

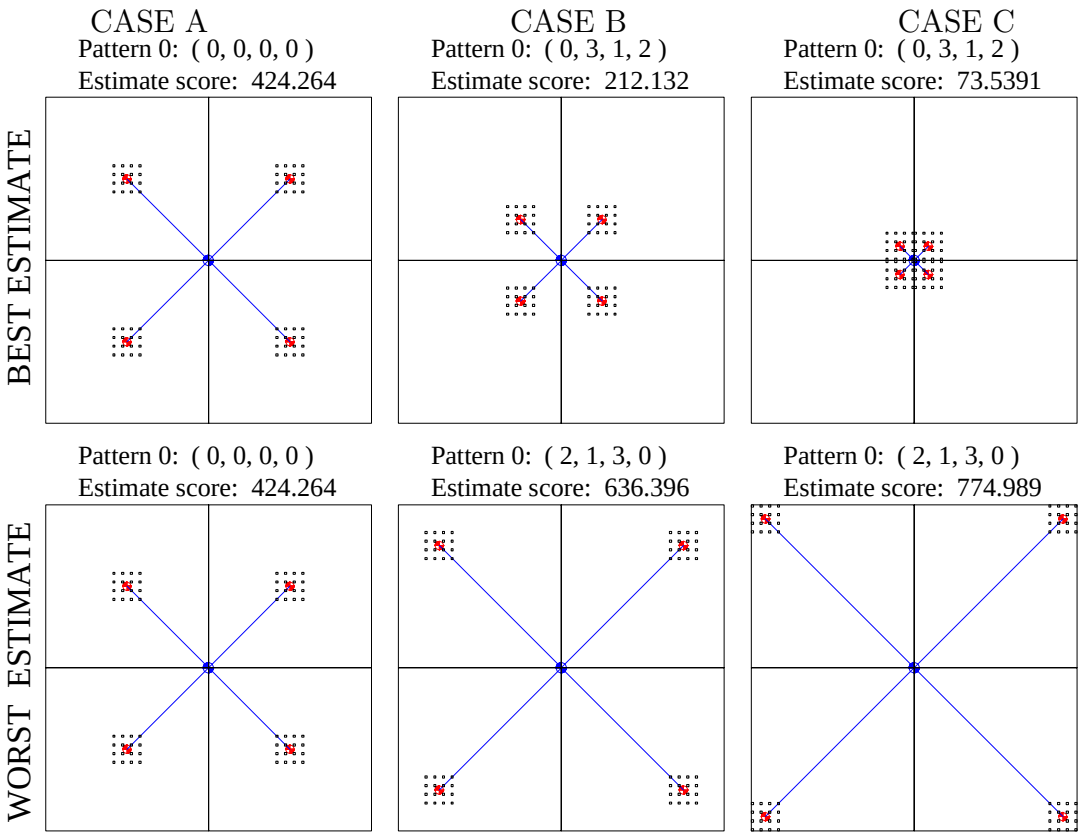


Figure 5.10: Best and Worst Estimator results for 4-PCB, RTHM experiment 2. Values in mm/component

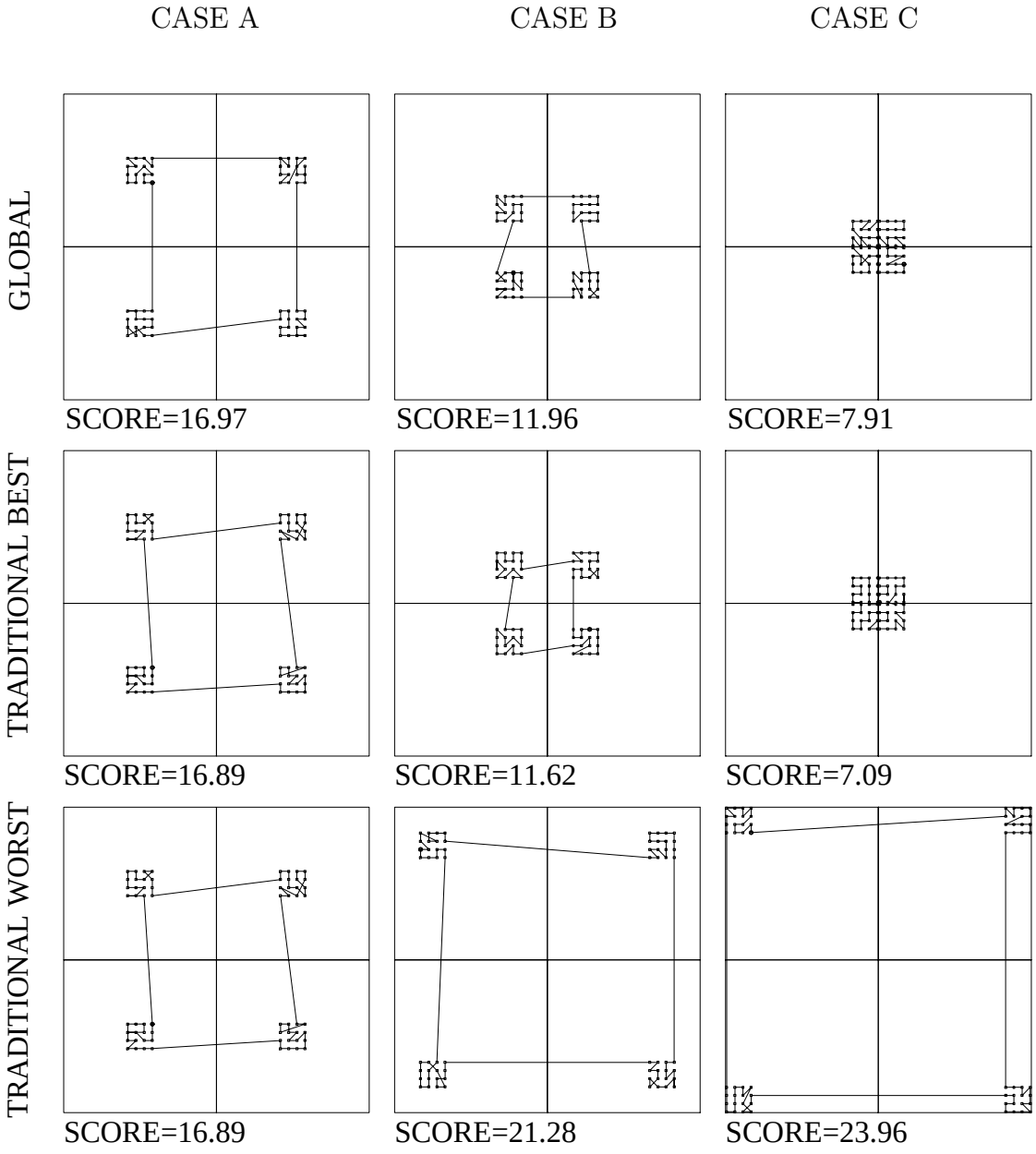


Figure 5.11: Solutions for AIM experiment 2 for 4-PCB. Scores in mm/component.

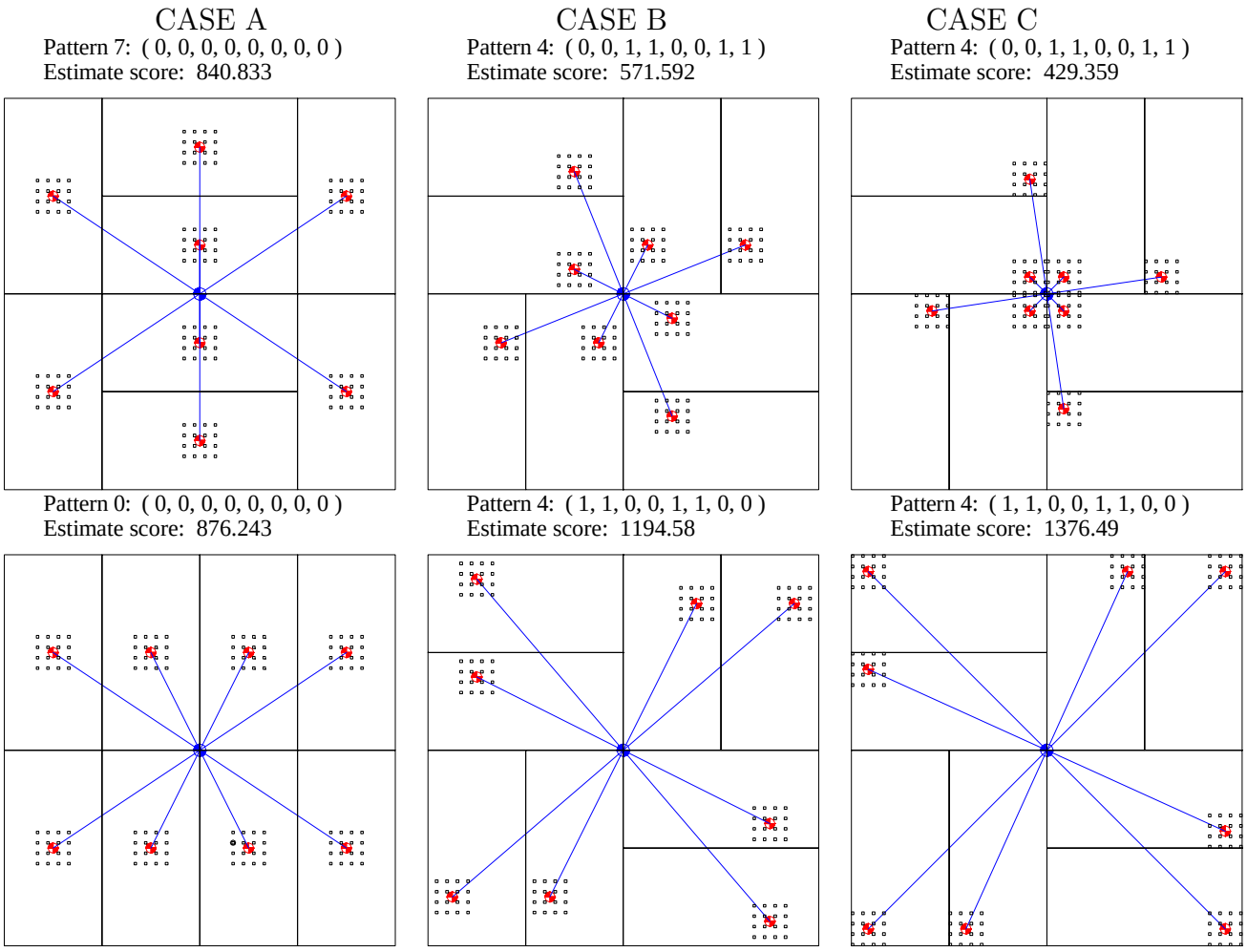


Figure 5.12: Best and Worst Estimator results for 8-PCB, AIM experiment 2. Values in mm/component

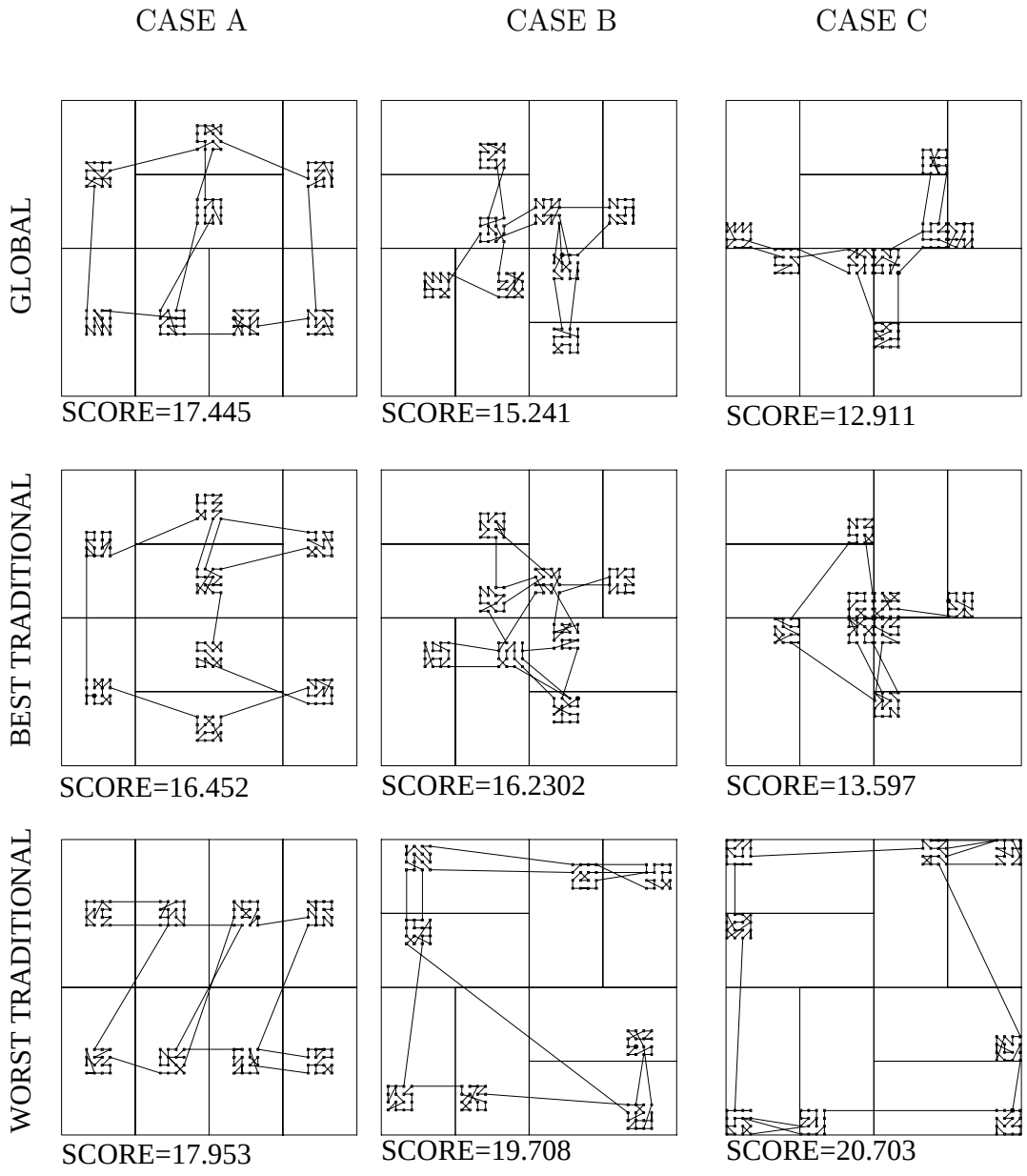


Figure 5.13: Solutions for AIM experiment 2 for 8-PCB. Scores in mm/component.

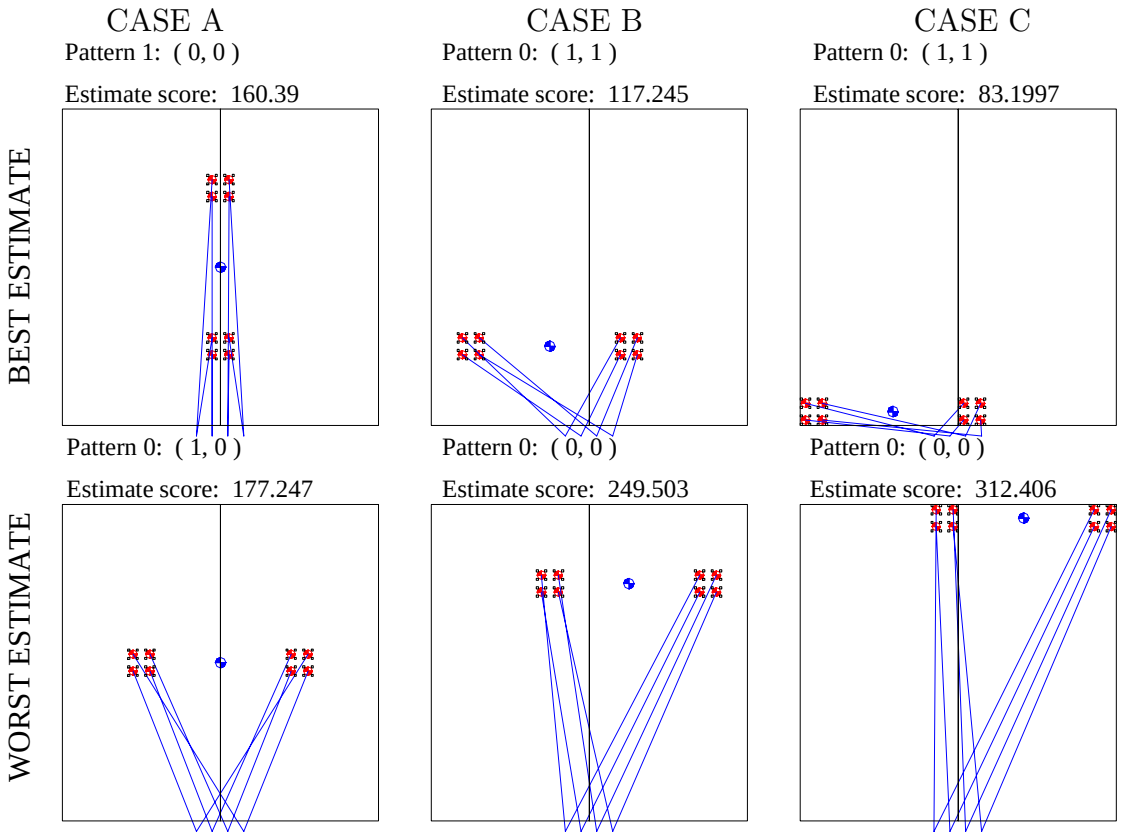


Figure 5.14: Best and Worst Estimator results for 2-PCB, PAPM experiment 2. Values in mm/component

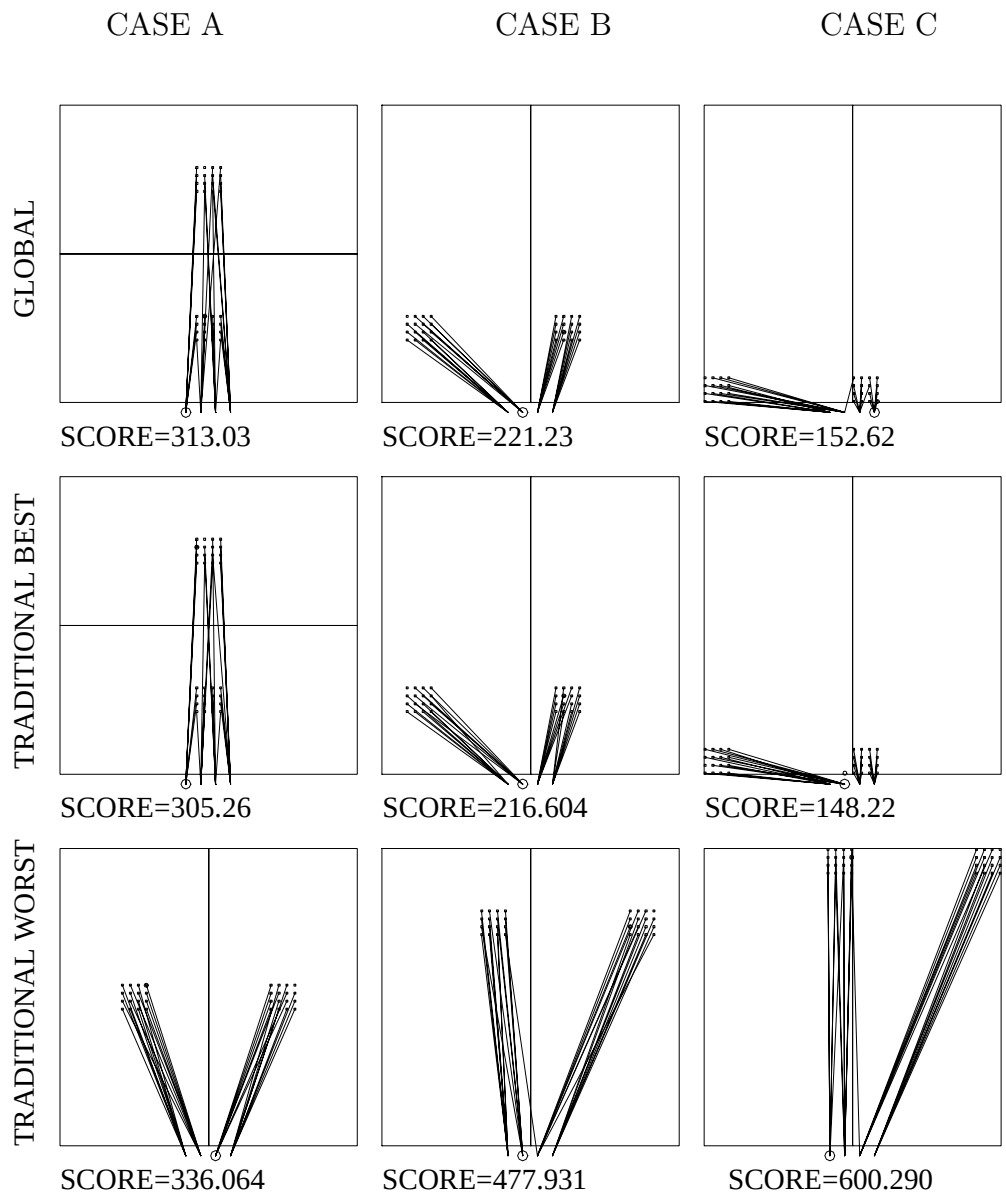


Figure 5.15: Solutions for PAPM experiment 2 for 2-PCB. Scores in mm/component.

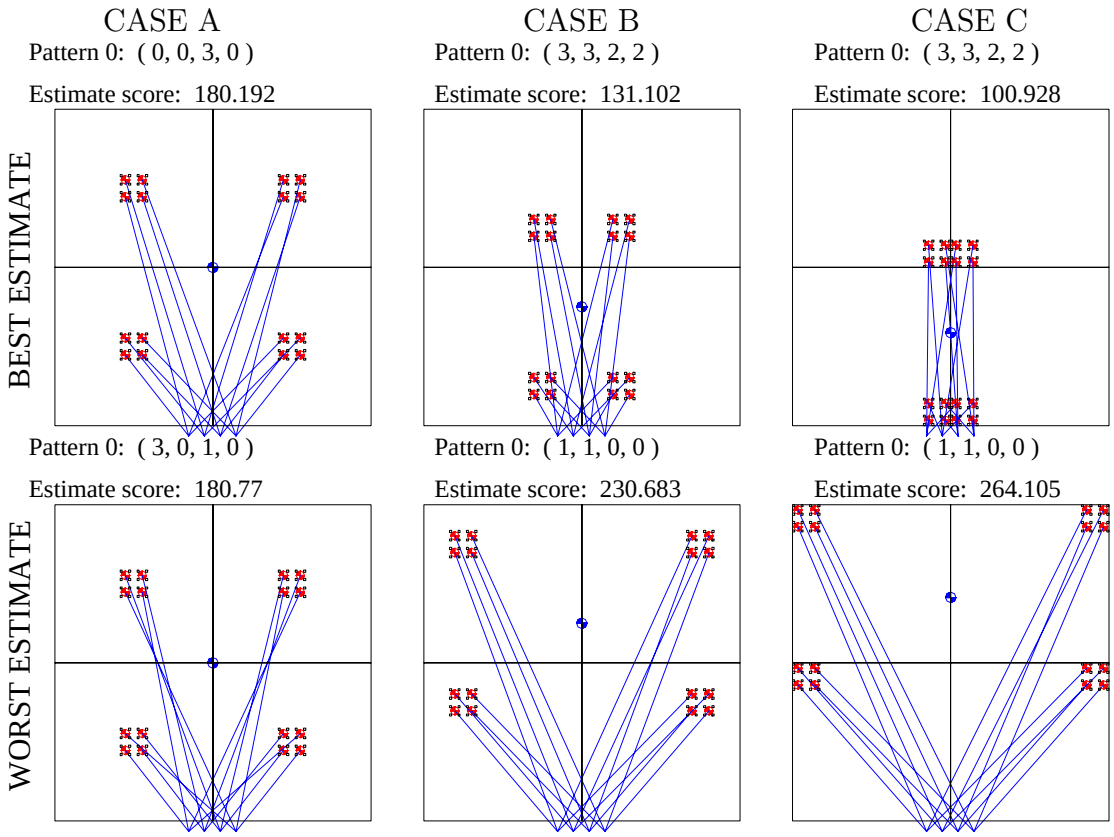


Figure 5.16: Best and Worst Estimator results for 4-PCB, PAPM experiment 2. Values in mm/component

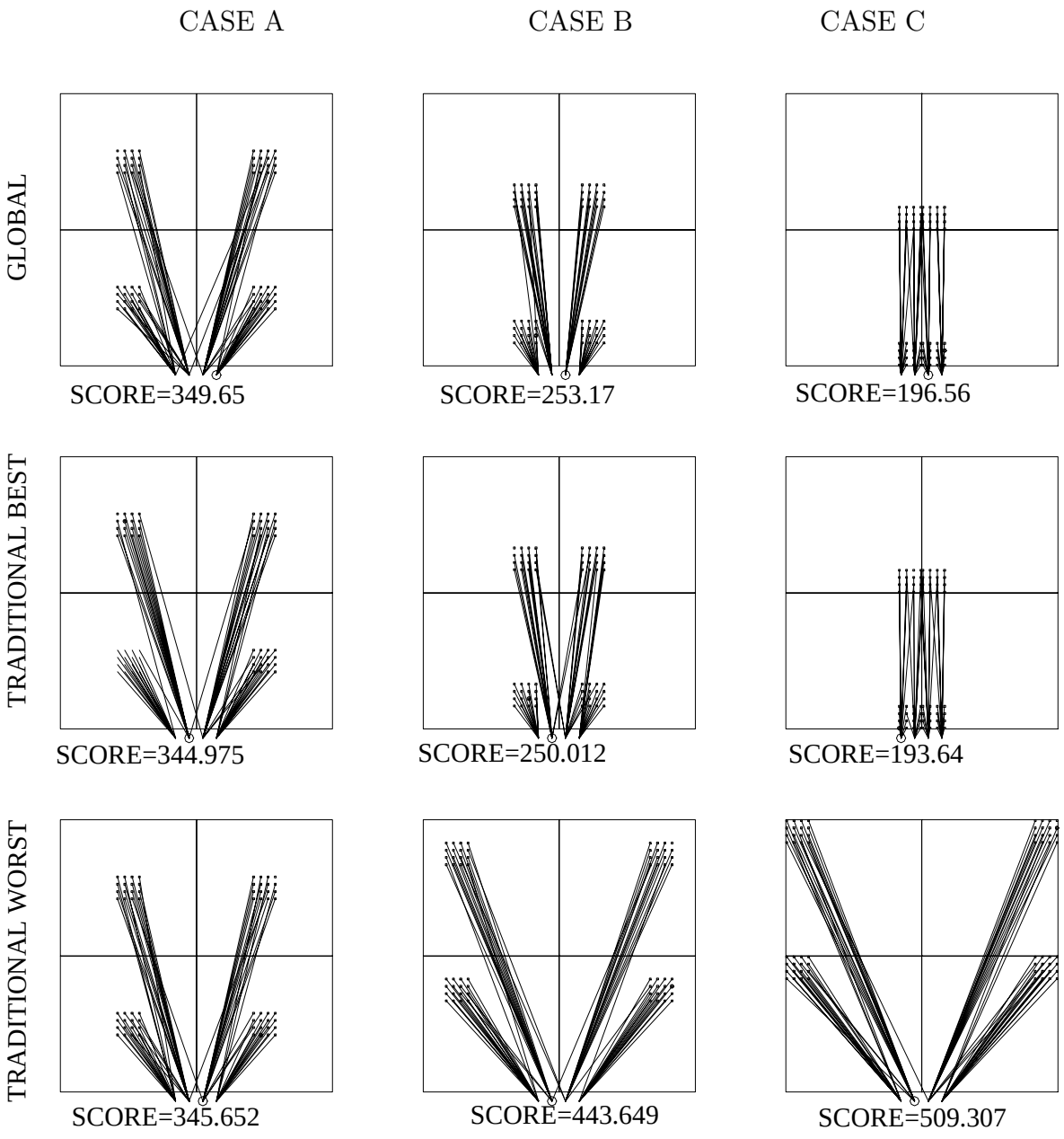


Figure 5.17: Solutions for PAPM experiment 2 for 4-PCB. Scores in mm/component

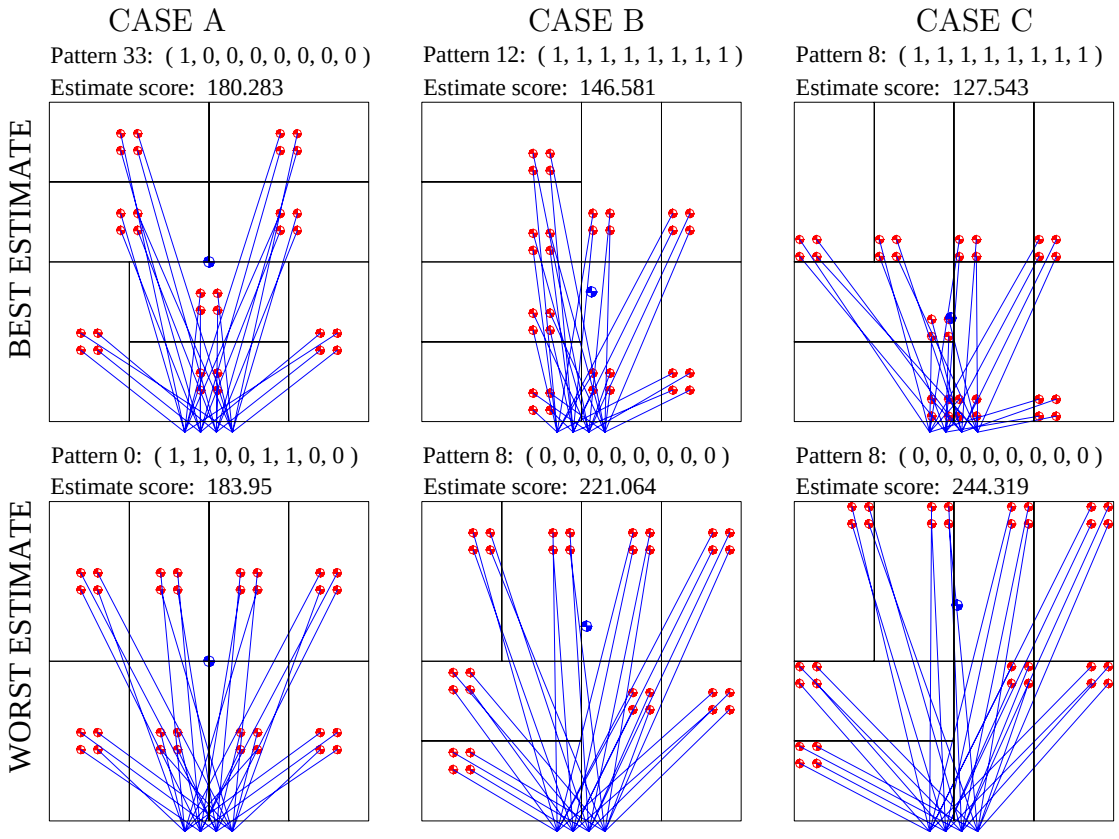


Figure 5.18: Best and Worst Estimator results for 8-PCB, PAPM experiment 2. Values in mm/component

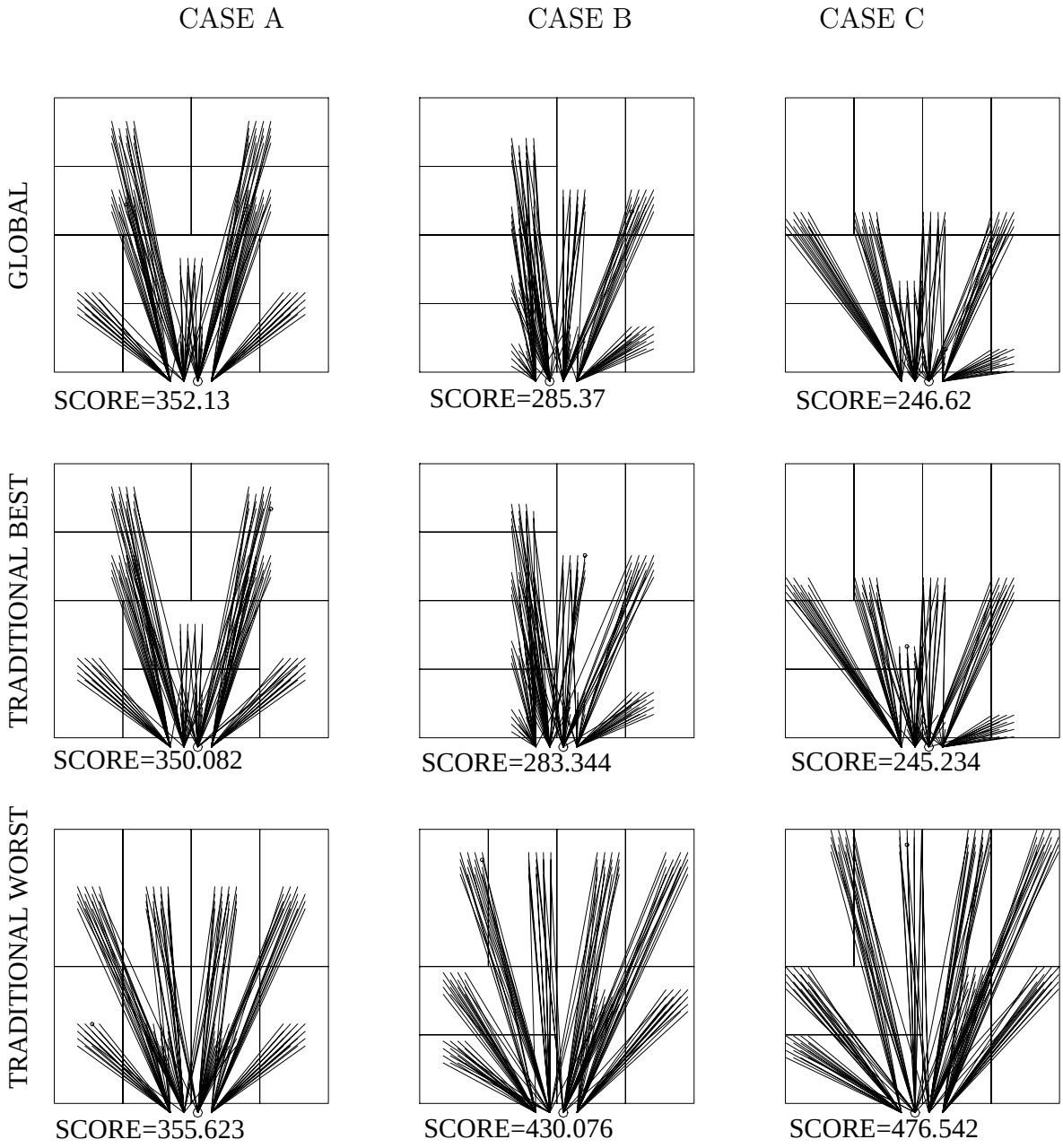


Figure 5.19: Solutions for PAPM experiment 2 for 8-PCB. Scores in mm/component.

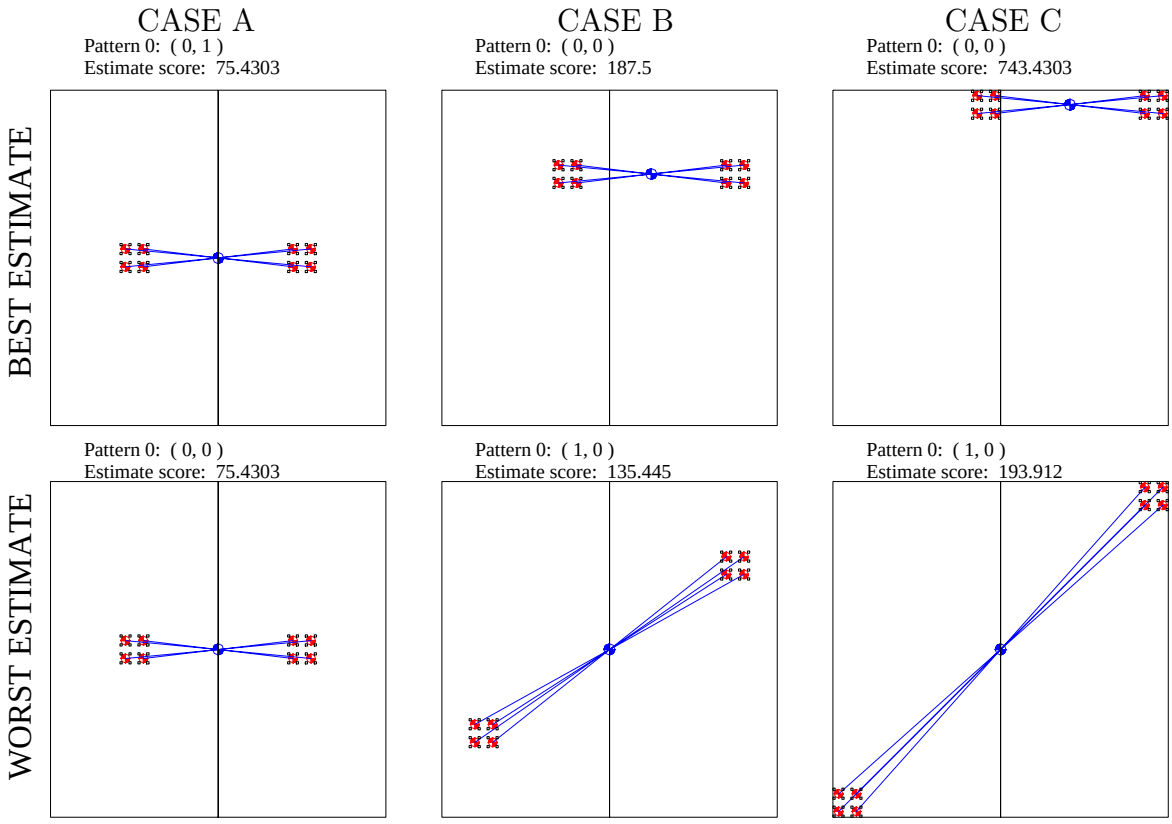


Figure 5.20: Best and Worst Estimator results for 2-PCB, RTHM experiment 2. Values in mm/component

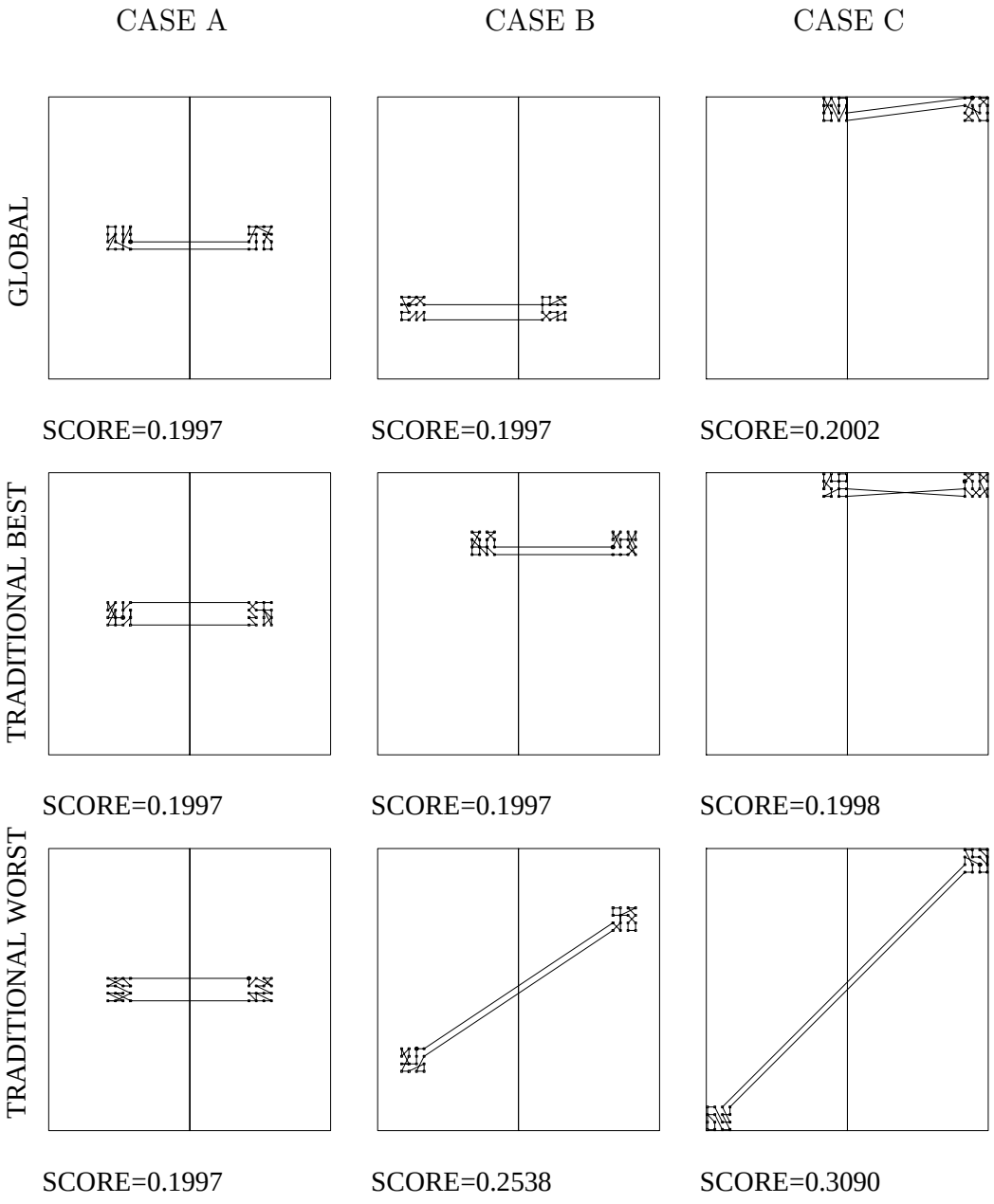


Figure 5.21: Solutions for RTHM experiment 2 for 2-PCB. Scores in sec/component.

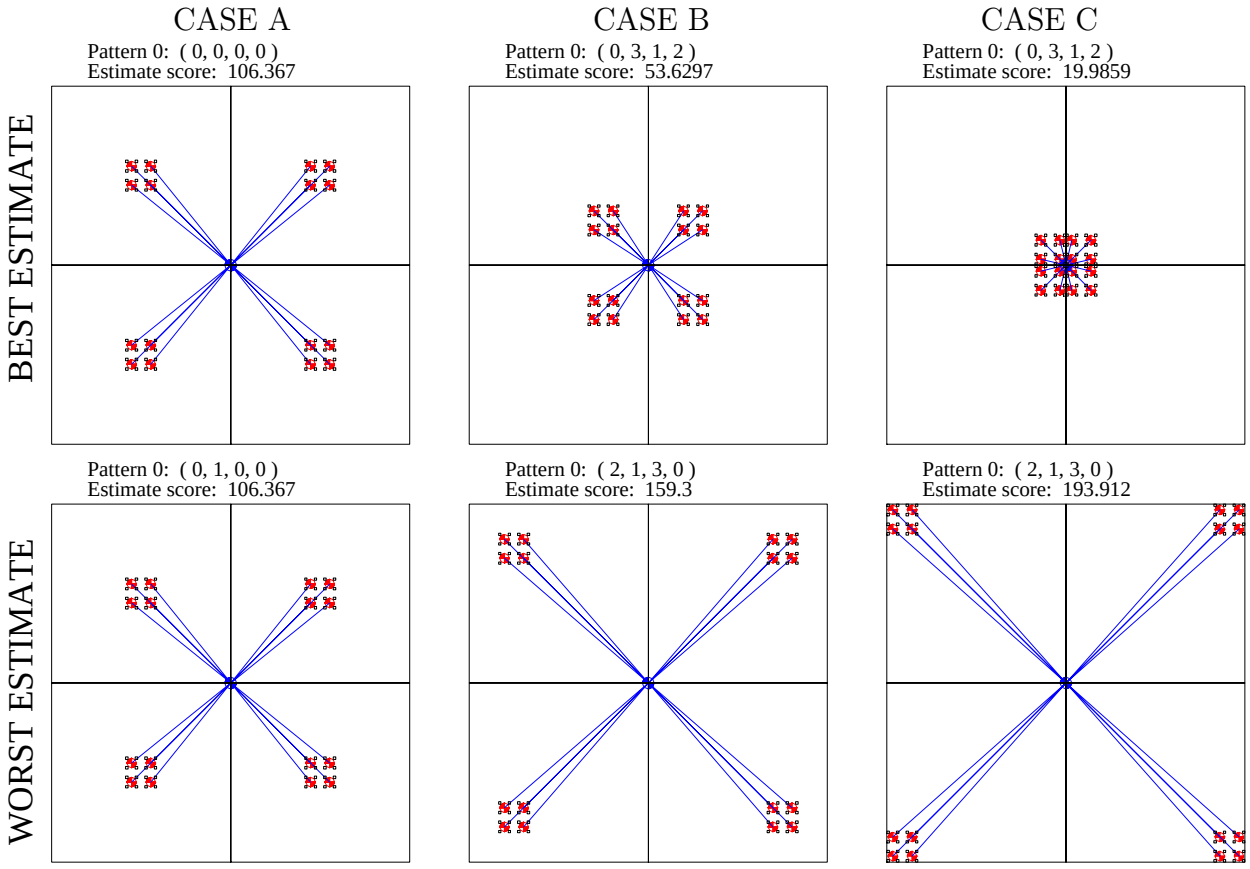


Figure 5.22: Best and Worst Estimator results for 4-PCB, RTHM experiment 2. Values in mm/component

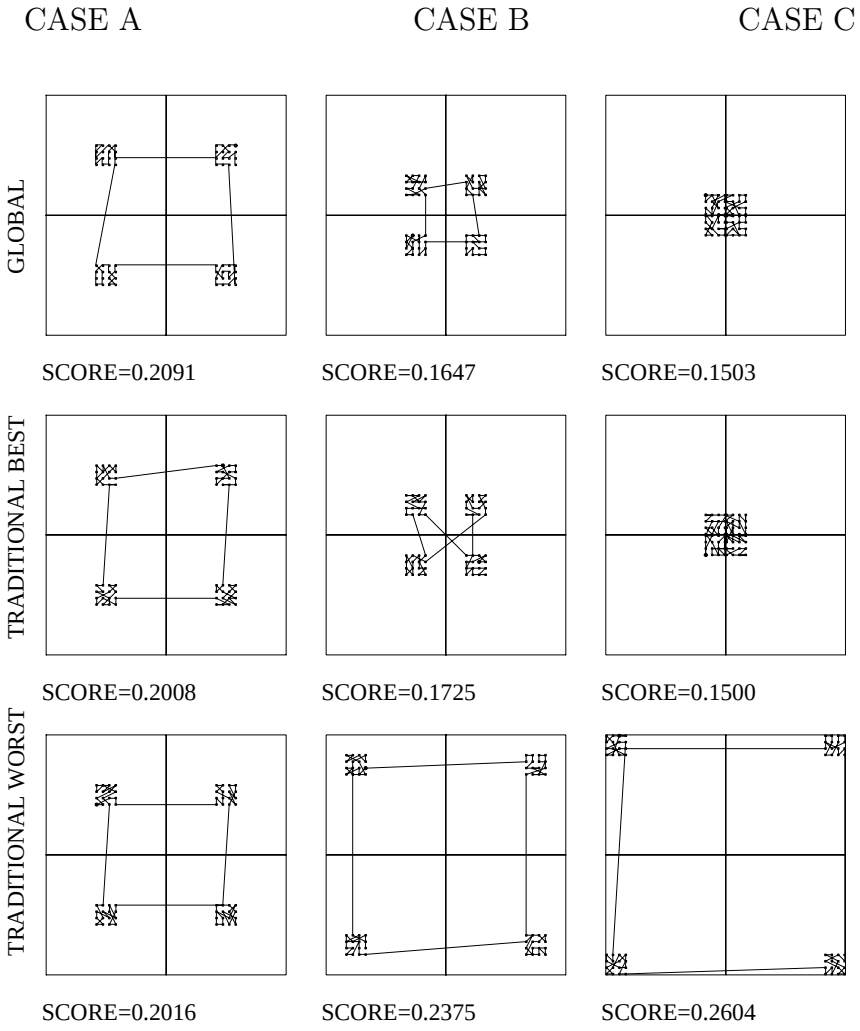


Figure 5.23: Solutions for RTHM experiment 2 for 4-PCB. Scores in sec/component.

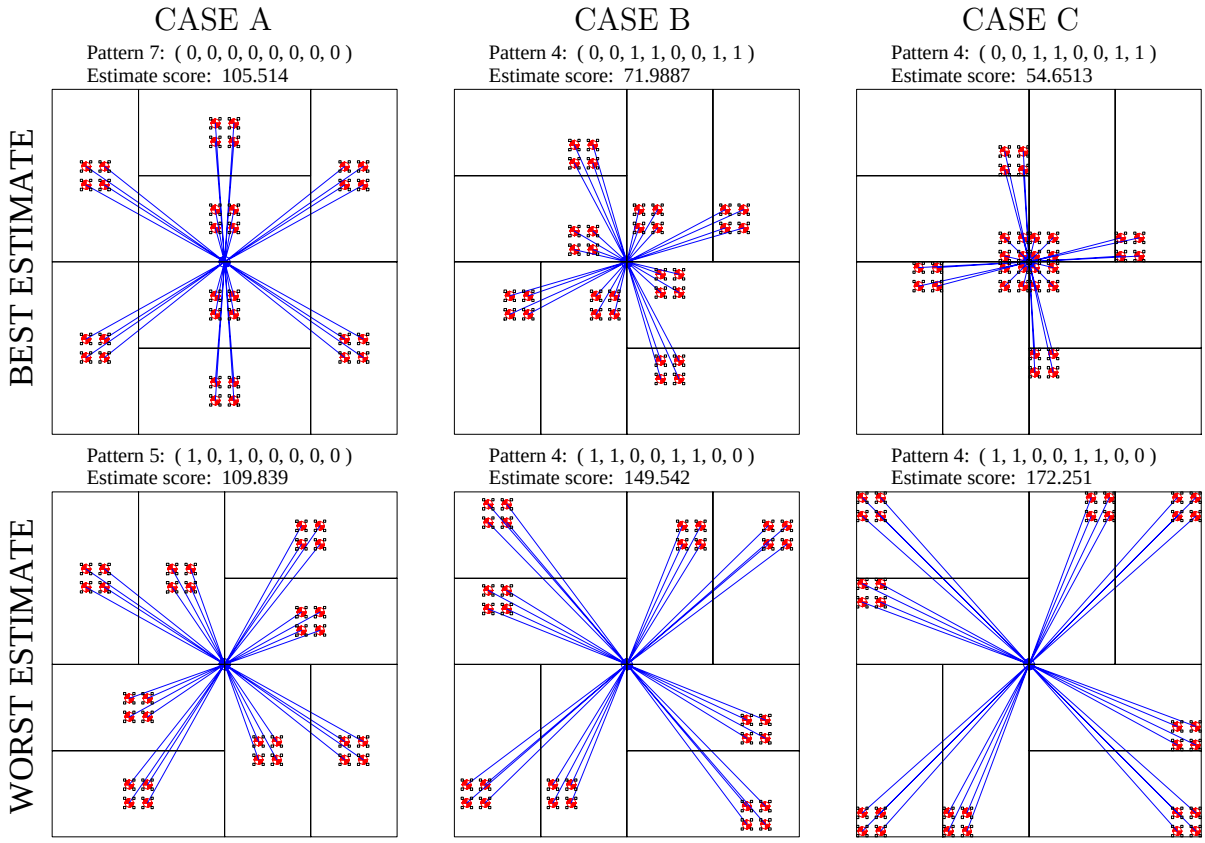


Figure 5.24: Best and Worst Estimator results for 8-PCB, RTHM experiment 2. Values in mm/component

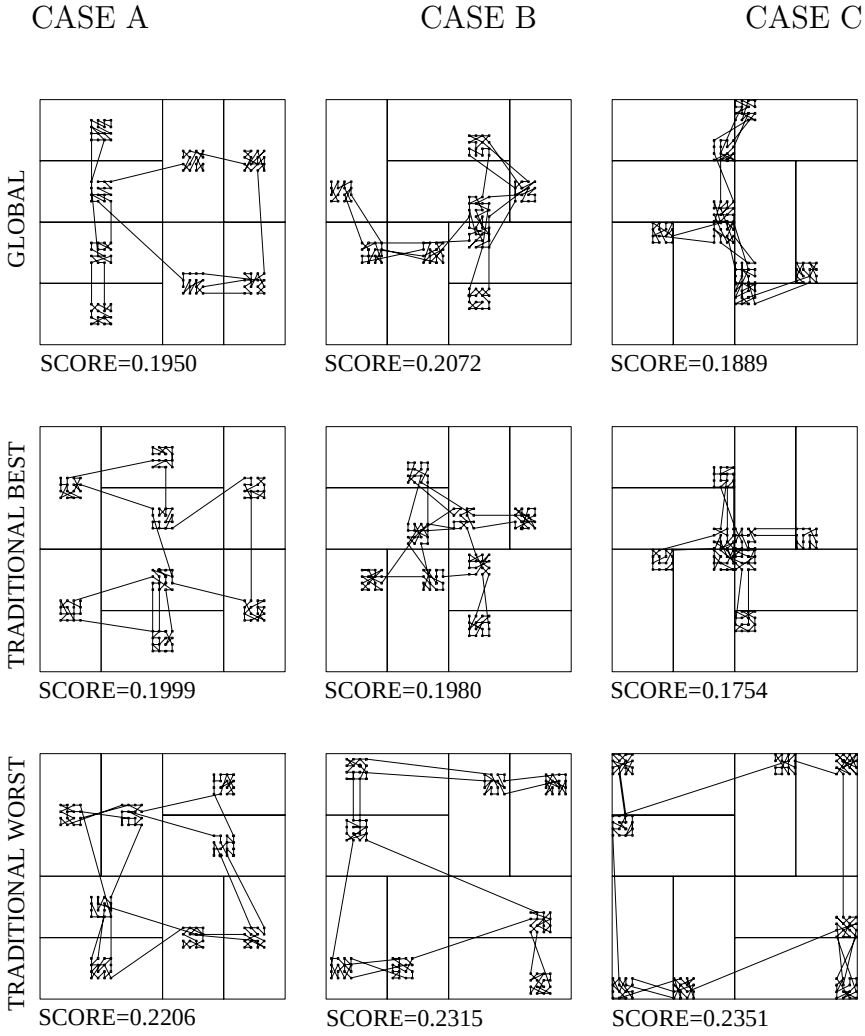


Figure 5.25: Solutions for RTHM experiment 2 for 8-PCB. Scores in sec/component.

5.3 Experiment 3 Results

A summary of all the data from experiment 3 is given in Table 5.3. The raw data and the panel designs resulting from these analyses are given in Appendix E. The graphical illustrations of the solutions are presented collectively for each machine type at the end of this section.

5.3.1 PAPM

Experiment 3, 2-PCB

As presented in section 4.6, this case was developed to separate component location eccentricity from component type eccentricity. The Estimated Best and Worst results are illustrated in Figure 5.26, and the solution sequences (unrequired moves only) are shown in Figure 5.27. The ELPI was 0.6%, suggesting that all the possible design alternatives can produce nearly the same placement time results. The TLPI and GLPI scores were within 1% of each other, supporting the likeness of the possible designs.

Experiment 3, 4-PCB

The Estimated Best and Worst results are illustrated in Figure 5.28, and the solution sequences (unrequired moves only) are shown in Figure 5.29. The ELPI was 0.8%, suggesting that all the possible design alternatives are equivalent candidates for the best possible placement time solution. Both the TLPI and GLPI magnitudes were less than 1%, supporting the assumption made due to the ELPI prediction. It is noted that the GLPI was negative. However, since the magnitude was below 1%, it was not considered unusual, the assumption being that such a low magnitude was associated with all panel designs having similar values for this experimental instance.

Table 5.3: Experiment 3 summary of results.

MACH	*No. of PCB*	*Estimated Worst*	*Estimated Best*	*Traditional Worst*	*Traditional Best*	*Global*	*ELPI*	*TLPI*	*GLPI*
PAPM	2	178.005	177.03	335.9	334.68	332.5	0.55%	0.36%	1.01%
	4	181.676	180.284	351.362	349.137	352.823	0.77%	0.63%	-0.42%
	8	184.854	180.582	361.636	361.957	362.377	2.31%	-0.09%	-0.20%
RTHM	2	77.67	76.35	0.2245	0.2167	0.2156	1.70%	3.47%	3.96%
	4	108.02	107.55	0.2244	0.2288	0.2231	0.44%	-1.96%	0.58%
	8	111.696	106.427	0.2359	0.2337	0.2332	4.72%	0.93%	1.14%

Experiment 3, 8-PCB

The Estimated Best and Worst results are illustrated in Figure 5.30, and the solution sequences (unrequired moves only) are shown in Figure 5.31. The ELPI was about 2%, indicating that there may be very little difference between all the possible design alternatives, in spite of the larger number design alternatives for the eight-PCB scenario. Both the TLPI and GLPI magnitudes were less than 1%, supporting the assumption made due to the ELPI prediction. It is noted here that both the TLPI and GLPI were negative. However, since their magnitudes were below 1%, it was not considered unusual, the assumption being that such low magnitudes were associated with all panel designs for this experimental instance having similar values.

The rationale behind the lack of a panelization advantage may be due to the panel taking on more uniform component distributions as the panel scenario increases in PCB population from two, four, and finally eight PCB per panel. The eccentricity associated with the component types distribution may be unable to compensate for the increasing uniformity of the component spacings across the PCB and panel layouts as the PCB boundaries become smaller as the PCB per panel parameter increases. As seen in experiment 2, case A for the AIM and PAPM, component distributions which are centered on the PCB centers yield low LPI with panelization implementation. It is apparent that for experiment 3 that the component type eccentricity is not sufficient to overcome the component location layout uniformity.

The PAPM experiment 2 showed strong support for differences in component placement times when panelization was considered. However, experiment 3 for the PAPM did not effectively show a major impact from the eccentricity of the component type distribution when the component location distribution is centered on the PCB geometric center, as well as being uniformly distributed across the panel.

5.3.2 RTHM

Experiment 3, 2-PCB

The Estimated Best and Worst results are illustrated in Figure 5.32, and the placement solution paths are shown in Figure 5.33.

The ELPI is 1.7%. The small difference here indicates virtually no advantage in any of the panel design alternatives. The TLPI and GLPI are 4%, which is slightly higher than the ELPI would suggest. There is also a 0.5% difference between the global and Traditional Best scores.

Experiment 3, 4-PCB

The Estimated Best and Worst are illustrated in Figure 5.34, and the placement solution sequences are shown in Figure 5.35.

The ELPI was 0.4%. The TLPI and GLPI are -2% less than 1%, respectively. The negative LPI has been addressed before for magnitudes less than 2%. The best panel design selected by the estimator function does not match that selected by the global analysis. These results indicate that there is little difference in any of the panel designs, given the scenario for experiment 3.

Experiment 3, 8-PCB

The Estimated Best and Worst results are illustrated in Figure 5.36, and the solution placement sequences are shown in Figure 5.37.

The ELPI, TLPI, and GLPI were 5%, 11% and 9%, respectively. The results from the RTHM Experiment 3 experiments seem to suggest that for a PCB layout with a uniform distribution of components and a non-uniform component type distribution, the consider-

ation of panelization will work marginally well only for the eight-PCB panel, where there are a greater number of panel designs from which to choose in the cycle time reduction process. However, it is of limited appeal, since the ELPI appears to have difficulty keeping a correlation with such an improvement.

The results from this experiment may reinforce the concept that the distribution of components is of greater impact to panelization than the component type distribution for the RTHM. This statement is made in the context of the experiment parameters established for these idealized panel designs in experiment 3.

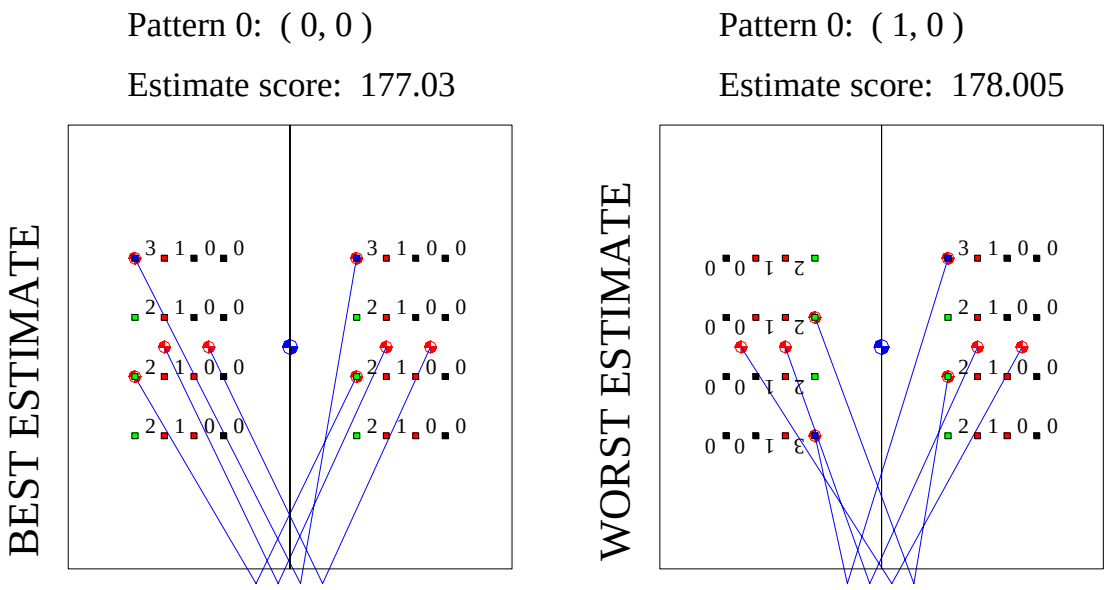


Figure 5.26: Best and Worst Estimator results for 2-PCB, PAPM experiment 3. Values in mm/component

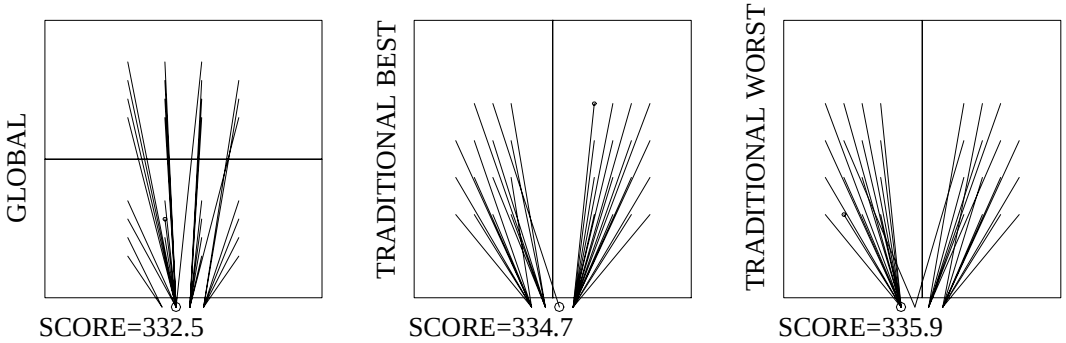


Figure 5.27: Solutions for PAPM experiment 3 for 2-PCB. Scores in mm/component.

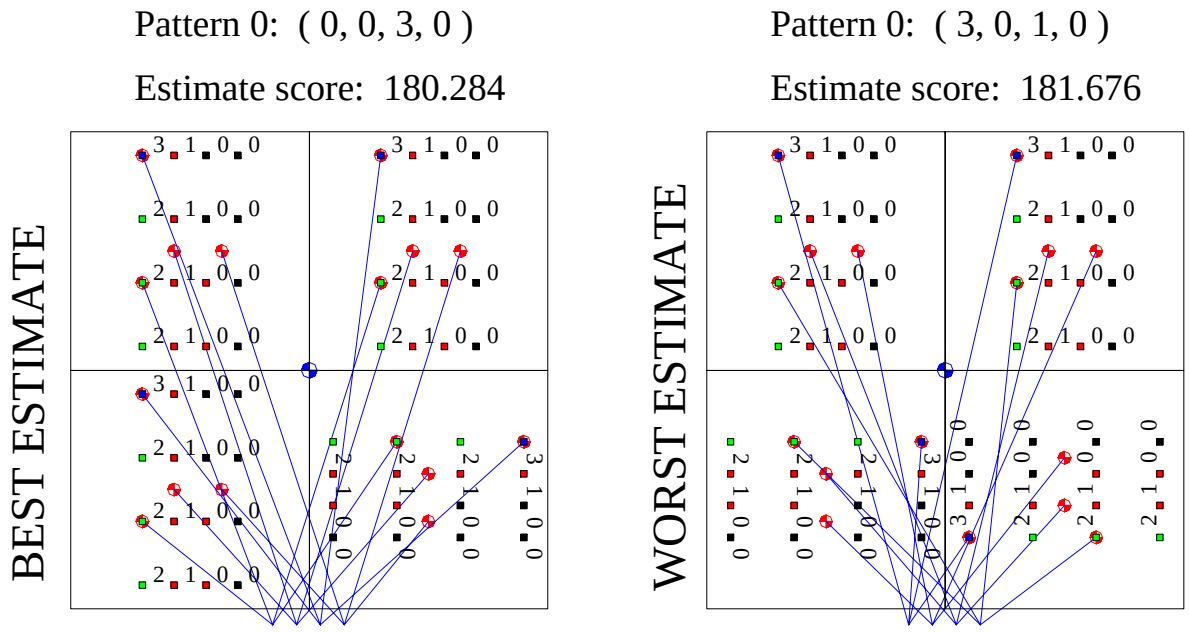


Figure 5.28: Best and Worst Estimator results for 4-PCB, PAPM experiment 3. Values in mm/component

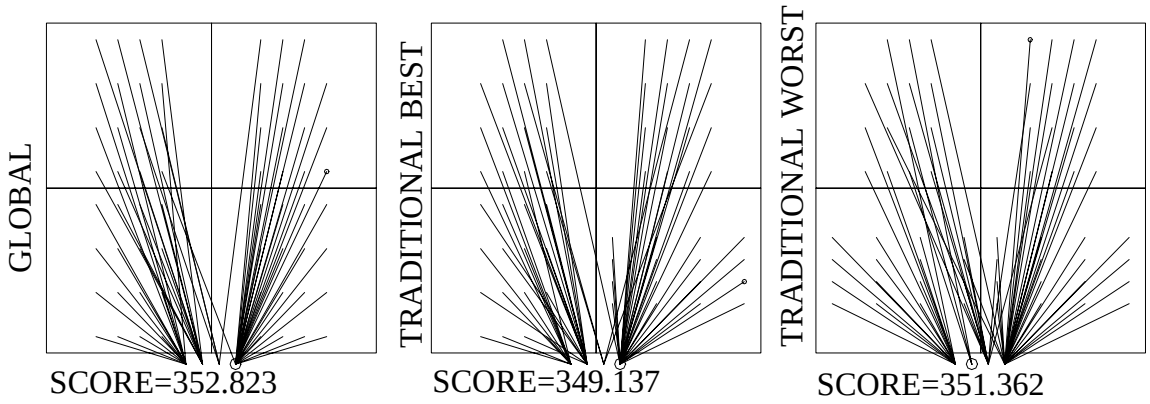


Figure 5.29: Solutions for PAPM experiment 3 for 4-PCB. Scores in mm/component.

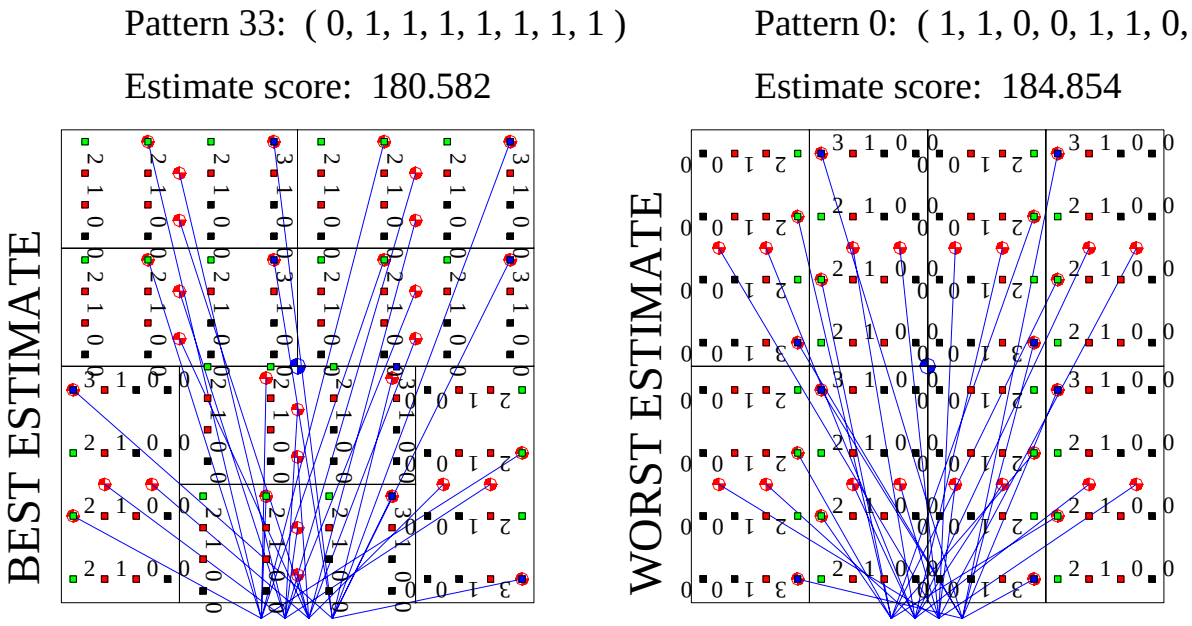


Figure 5.30: Best and Worst Estimator results for 8-PCB, PAPM experiment 3. Values in mm/component

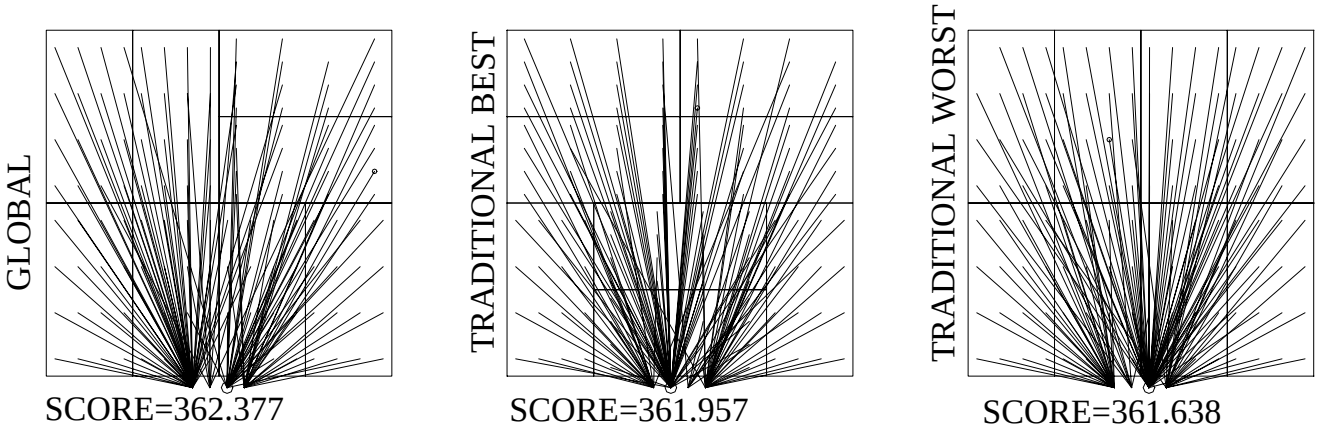


Figure 5.31: Solutions for PAPM experiment 3 for 8-PCB. Scores in mm/component.

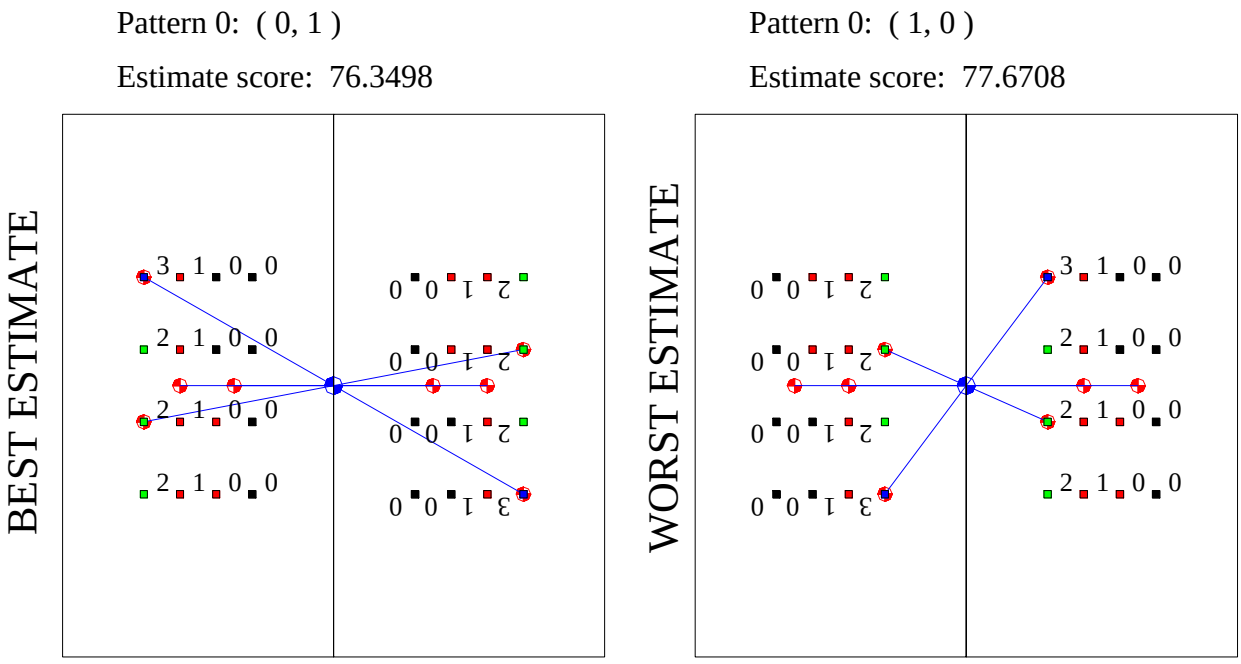


Figure 5.32: Best and Worst Estimator results for 2-PCB, RTHM experiment 3. Values in mm/component

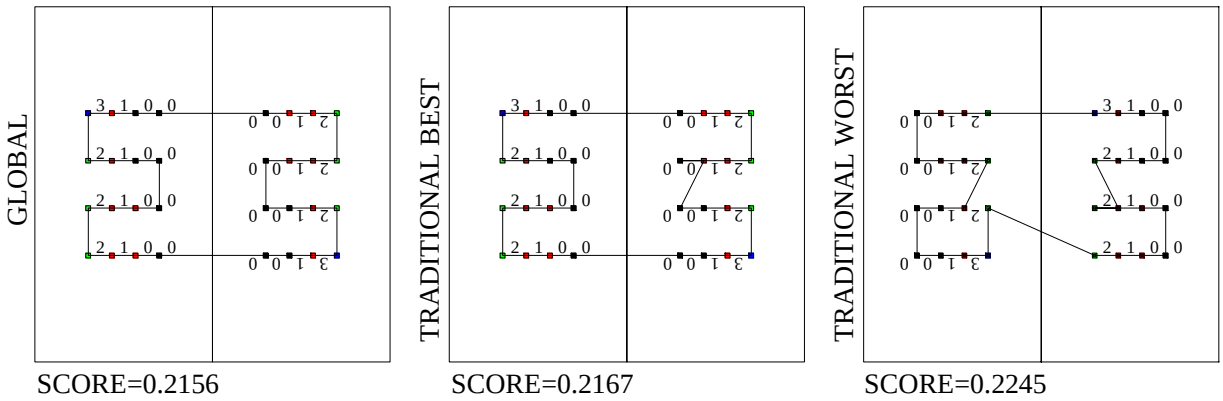


Figure 5.33: Solutions for RTHM Experiment 3 for 2-PCB. Scores in mm/component.

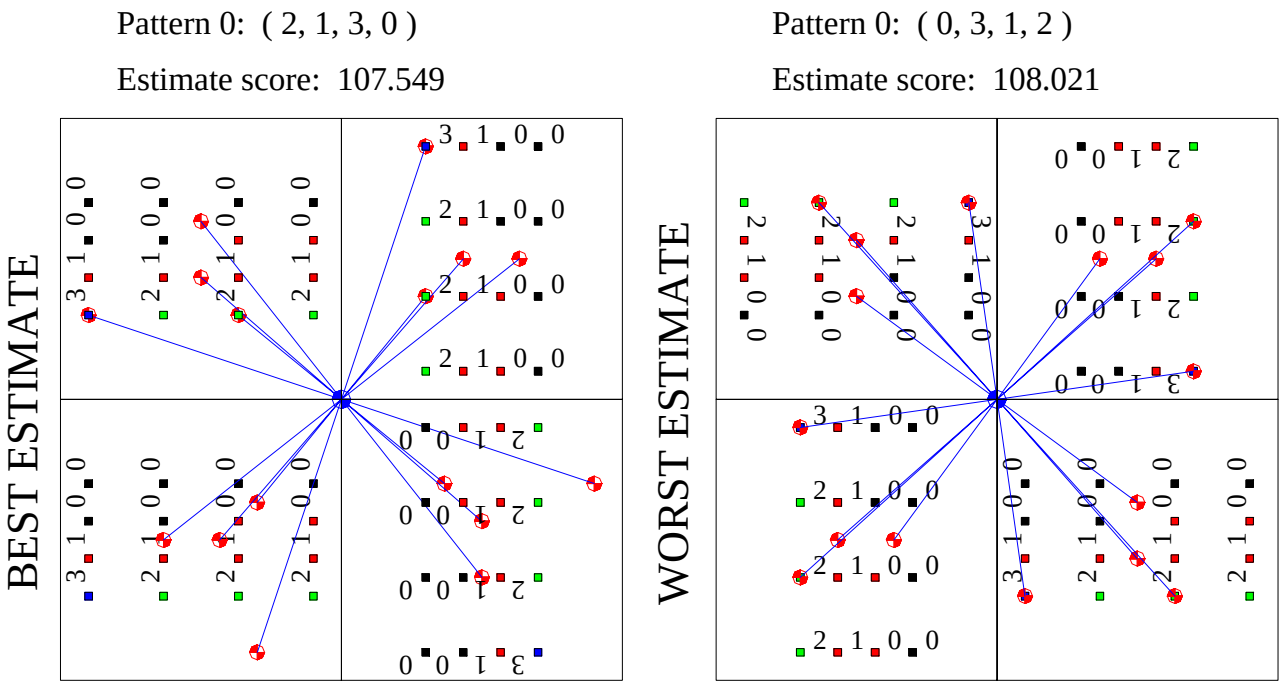


Figure 5.34: Best and Worst Estimator results for 4-PCB, RTHM experiment 3. Values in mm/component

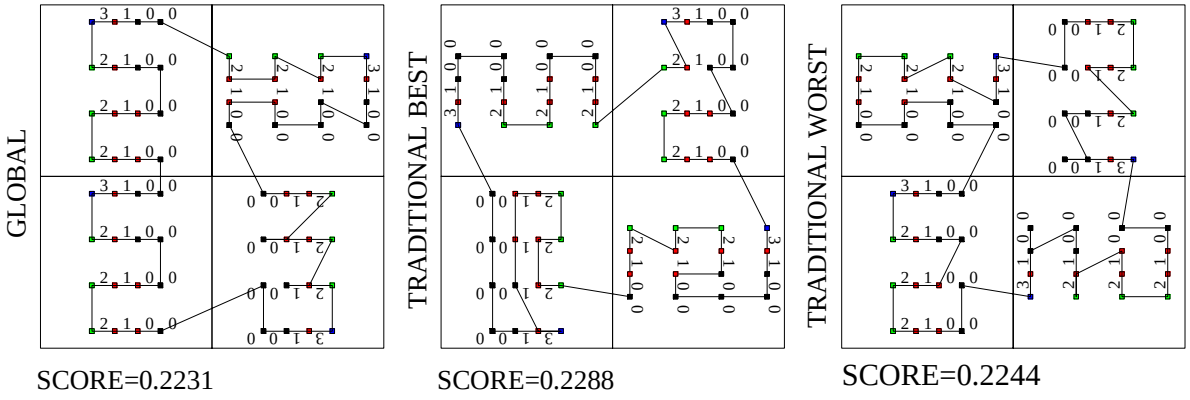


Figure 5.35: Solutions for RTHM Experiment 3 for 4-PCB. Scores in mm/component.

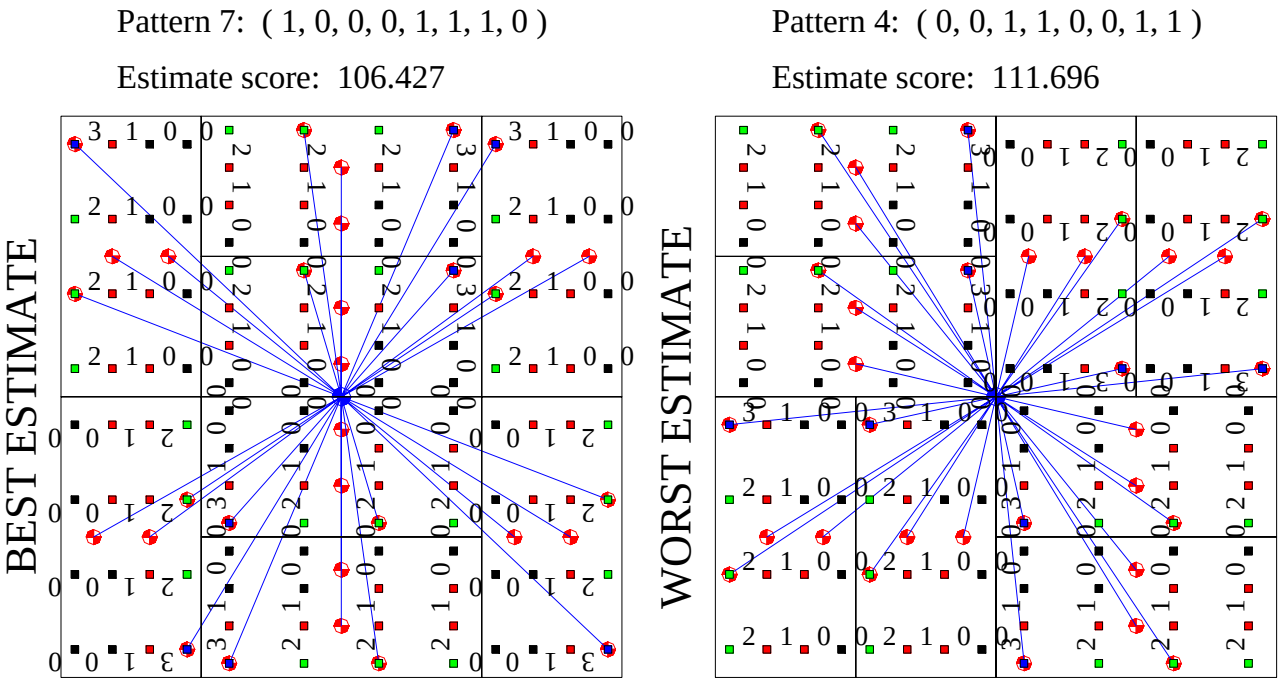


Figure 5.36: Best and Worst Estimator results for 8-PCB, RTHM experiment 3. Values in mm/component

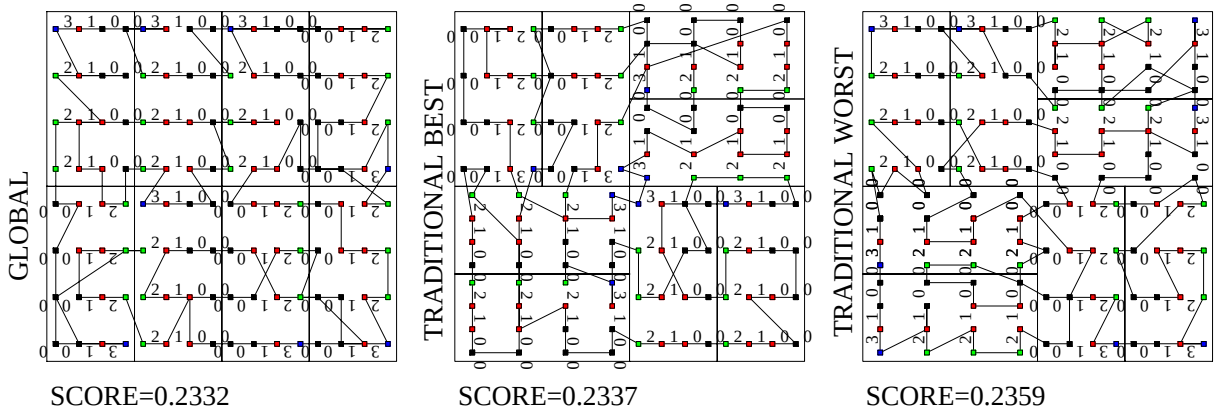


Figure 5.37: Solutions for RTHM Experiment 3 for 8-PCB. Scores in mm/component.

5.4 Experiment 4 Results

A summary of all the data from experiment 4 is given in Tables 5.4 and 5.5. Two tables are used in lieu of the single table from earlier summaries only for the purpose of facilitating the page layout.

Table 5.4: Experiment 4 summary of results.

MACH	*No. of PCB*	*Estimated Worst*	*Estimated Best*	*Traditional Worst*	*Traditional Best*	*Global*
PAPM	8	182.44	177.695	346.031	324.612	326.273
RTHM	8	82.6557	72.7494	0.4489	0.4412	0.4309

Table 5.5: Experiment 4 summary of results (LPI).

MACH	*No. of PCB*	*ELPI*	*TLPI*	*GLPI*
PAPM	8	2.60%	6.19%	5.71%
RTHM	8	11.99%	1.72%	4.01%

5.4.1 PAPM

The Estimated Best and Worst panel designs are illustrated in Figure 5.38, where the component type geometric centers are displayed. The panel layout has 216 components; the placement sequence path illustrations are cluttered and not very useful. Thus, they will be omitted in the solution displayed graphically in Figure 5.39.

The ELPI is 3%, with the TLPI and GLPI being 6% for both. It is of note that the panel design resulting from the global approach solution matches that of the Traditional Best solution, and the scores are within 0.5%. Though the estimator span was small, this match influences the idea that a particular panel design is favored over others for this particular machine type. The difference between the best and worst solutions is 5.7%. This value,

combined with the above observations, leads one to conclude that this case would show a small, though real, LPI if panelization were considered.

5.4.2 RTHM

The Estimated Best and Worst panel designs are illustrated in Figure 5.40. The two selected panel designs of Figure 5.40 have the same pattern, but their PCB rotations are opposite (180^o rotation) from each other. The ELPI is 12%.

The solutions for the RTHM situation are illustrated in Figure 5.41. The TLPI and GLPI are 2% and 4%, respectively. The global, Traditional Best and Traditional Worst panel design are all different. It is likely that there are several, nearly equal panel design alternatives. This situation would result in the global analysis settling at a design different from the one suggested from the estimator analysis. The problem in this experiment is that the global and the Traditional Best do not have similar score results. The difference between these two scores is nearly 5%, thus putting into doubt any benefit (as suggested by the TLPI) obtained from the analyses due to consideration of panelization in this experiment.

This industrial example illustrates that a difficult design situation is possible in applications. The number of component types (20) nearly matches the number of components (27) for each PCB. Consequently, there is a potential for a very large time penalty if two subsequent component placements of different component types had these types allocated at the opposite ends of the feeder bank. In such an event, the time required to travel from the first slot to the last would be (for these particular experiments) 0.15 seconds $\times$ 19, or 2.85 seconds. Given the table velocity of 133.3 millimeters/second, any table motion less than 46.8 millimeters would not penalize long table moves between placements. This distance is more than half of the shorter PCB dimension; the shorter dimension is also the axis along which most of the components are aligned. Thus, many apparently long moves produced from a solution would not actually be penalizing the cycle time once one accounts for the component allocation.

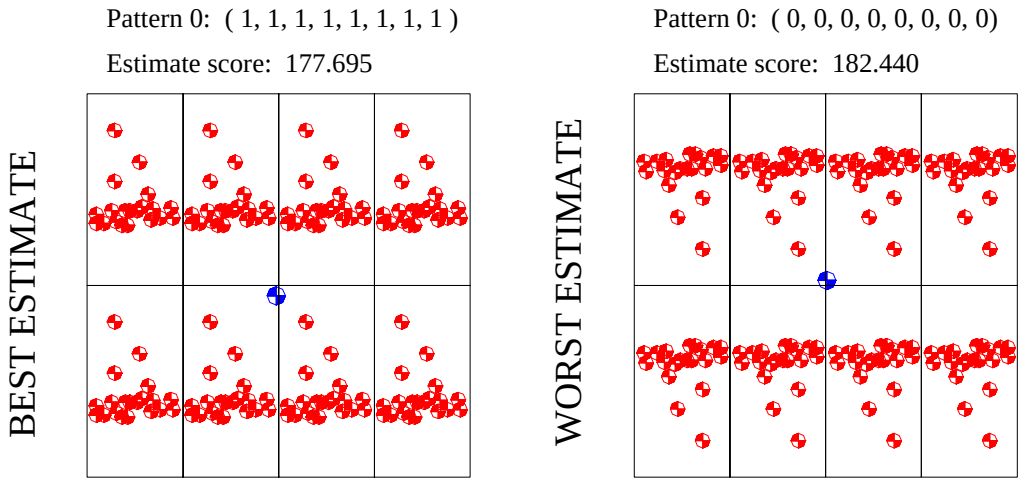


Figure 5.38: Best and Worst Estimator results for 8-PCB, PAPM experiment 4. Values in mm/component

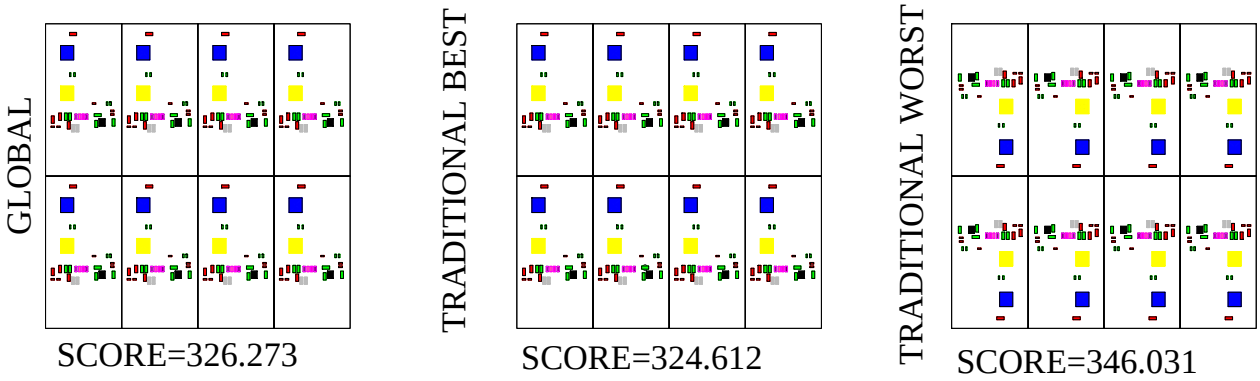


Figure 5.39: Solutions for PAPM experiment 4 for 8-PCB. Scores in mm/component.

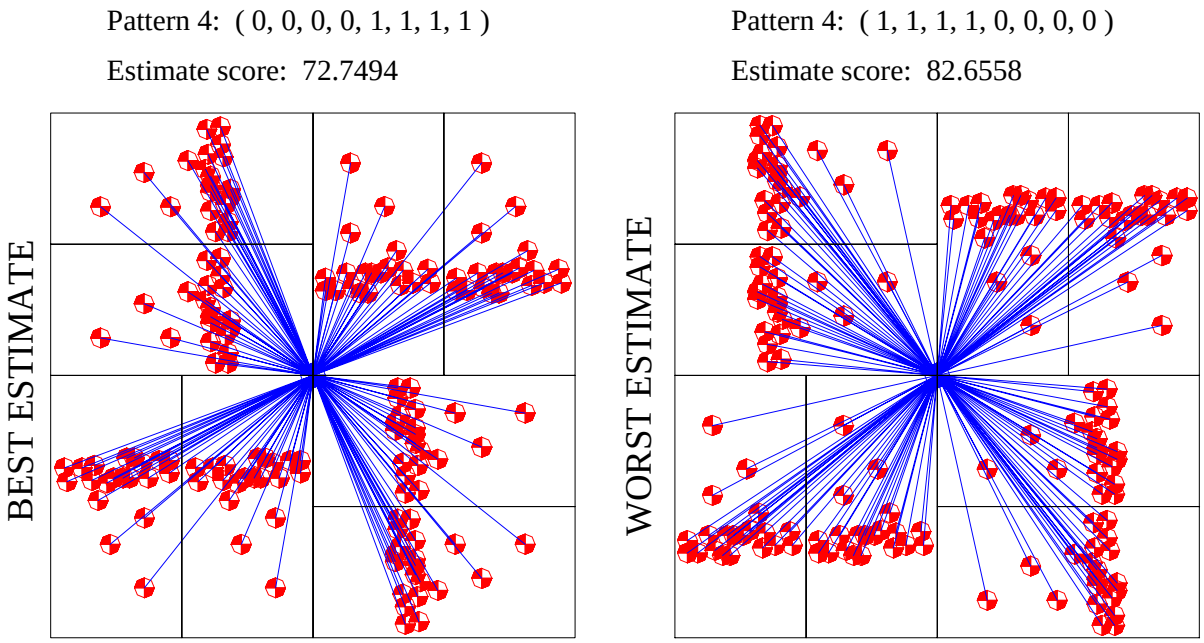


Figure 5.40: Best and Worst Estimator results for 8-PCB, experiment 4. Values in mm/component

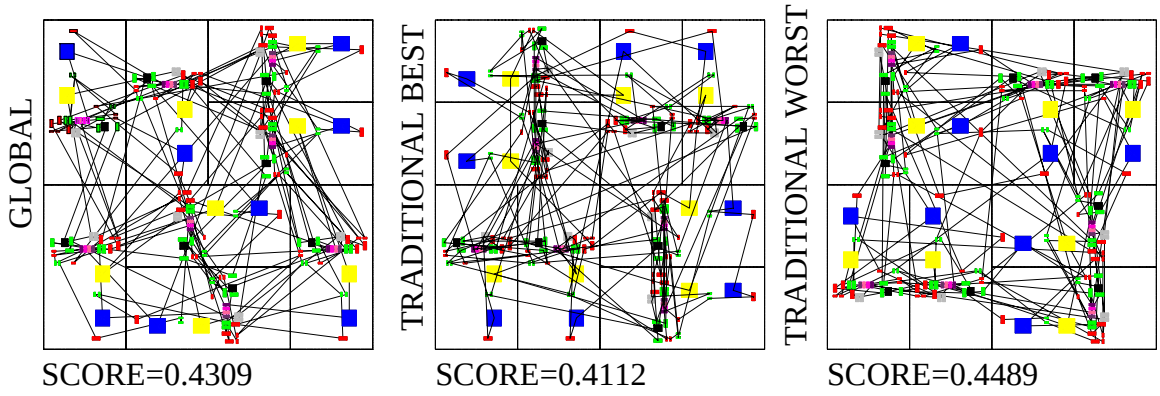


Figure 5.41: Solutions for RTHM experiment 4 for 8-PCB. Scores in mm/component.

5.5 Experiment 5 Results

A summary of the ELPI, TLPI and GLPI from experiment 5 is given in Tables 5.6, 5.7 and 5.8. Three tables are used in lieu of a single table only for the purpose of facilitating the page layout. Also, the average results for each machine type and panel scenario are given in Table 5.9 and the correlation between the ELPI and the TLPI and GLPI are given in Table 5.10. The raw data is presented in Appendix G. As first discussed in section 4.8, no graphical illustrations will be presented for experiment 5, due to the sheer number of different designs generated.

Table 5.6: Experiment 5 summary results, AIM.

	2 PCB			**4 PCB**			**8 PCB**		
MACH	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*
AIM	13.90%	13.33%	2.18%	20.25%	17.09%	12.86%	21.11%	5.45%	2.14%
	6.38%	10.14%	2.59%	21.22%	10.43%	4.36%	26.35%	28.45%	23.03%
	10.36%	8.31%	0.78%	16.65%	28.43%	24.39%	0.65%	15.17%	16.18%
	18.39%	15.88%	2.74%	24.10%	16.92%	12.37%	12.90%	17.53%	8.63%
	1.55%	5.52%	-2.13%	17.97%	-1.37%	2.47%	19.98%	19.74%	2.53%
	18.21%	7.78%	6.34%	6.50%	13.48%	17.81%	7.64%	7.47%	11.19%
	19.99%	0.26%	2.63%	3.91%	18.28%	13.70%	19.39%	5.65%	14.55%
	14.40%	2.11%	9.76%	10.67%	10.09%	8.93%	23.47%	19.86%	18.95%
	10.54%	-1.14%	1.22%	21.96%	13.30%	11.59%	25.66%	15.05%	15.14%
	7.00%	12.20%	7.13%	23.35%	14.38%	7.28%	17.09%	16.35%	12.95%

Table 5.7: Experiment 5 summary results, PAPM.

	2 PCB			**4 PCB**			**8 PCB**		
MACH	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*
PAPM	5.96%	2.73%	4.20%	4.56%	5.22%	5.63%	8.49%	7.79%	8.20%
	8.65%	3.54%	4.21%	5.69%	5.40%	5.54%	5.69%	5.03%	5.45%
	7.49%	3.03%	3.05%	5.18%	4.69%	5.01%	7.47%	7.41%	6.88%
	7.49%	8.99%	9.27%	2.55%	1.59%	1.88%	9.34%	6.74%	9.86%
	6.00%	1.82%	2.13%	4.94%	5.02%	5.49%	4.29%	2.49%	2.74%
	5.86%	0.42%	1.36%	3.45%	3.69%	4.19%	6.70%	6.15%	6.90%
	8.09%	3.45%	3.88%	6.20%	5.92%	6.21%	4.96%	4.38%	4.92%
	6.26%	1.75%	2.00%	5.18%	5.67%	5.68%	6.50%	5.76%	6.12%
	14.17%	10.28%	10.36%	10.65%	4.81%	10.53%	10.20%	9.31%	9.69%
	11.35%	6.25%	6.57%	19.21%	8.65%	17.81%	10.41%	9.96%	10.42%

Table 5.8: Experiment 5 summary results, RTHM.

	2 PCB			**4 PCB**			**8 PCB**		
MACH	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*	*ELPI*	*TLPI*	*GLPI*
RTHM	7.41%	-2.88%	0.93%	12.34%	1.32%	4.81%	13.87%	5.06%	10.62%
	11.33%	13.00%	11.60%	11.82%	7.30%	-0.06%	10.45%	11.87%	9.24%
	11.44%	-3.94%	0.37%	8.66%	-13.68%	0.40%	13.44%	-3.02%	0.29%
	23.99%	5.68%	-5.39%	2.79%	-12.07%	-11.28%	14.93%	11.91%	12.58%
	4.28%	1.52%	3.17%	9.74%	-0.86%	-4.85%	8.66%	9.99%	7.36%
	0.61%	-0.84%	-8.81%	8.68%	10.50%	9.32%	11.62%	2.64%	12.11%
	7.99%	-1.90%	-0.12%	9.32%	-8.15%	-3.03%	10.99%	12.65%	8.64%
	5.05%	13.05%	15.67%	8.61%	-1.30%	10.85%	11.96%	4.55%	3.98%
	19.04%	-13.02%	-18.58%	14.71%	3.00%	-6.38%	13.72%	-1.13%	-4.08%
	15.52%	1.92%	5.21%	33.03%	8.32%	12.38%	10.92%	12.86%	11.01%

Table 5.9: Experiment 5 summary *average* results with Confidence Intervals (CI).

MACH	*No. of PCB*	*Avg ELPI*	*Avg TLPI*	*Std Dev TLPI*	*Avg GLPI*	*Std dev GLPI*	*Lower CI (TLPI)*	*Upper CI (TLPI)*	*Lower CI (GLPI)*	*Upper CI (GLPI)*
AIM	2	12.07%	7.44%	5.72%	3.32%	3.47%	-0.47%	15.35%	-1.47%	8.12%
	4	16.66%	14.10%	7.53%	11.58%	6.42%	3.69%	24.51%	2.70%	20.45%
	8	17.42%	15.07%	7.23%	12.53%	6.67%	5.07%	25.07%	3.30%	21.76%
PAPM	2	8.13%	4.22%	3.24%	4.70%	3.08%	-0.26%	8.71%	0.45%	8.96%
	4	5.07%	6.80%	1.77%	7.40%	4.42%	4.35%	9.24%	1.29%	13.52%
	8	7.40%	6.50%	2.25%	7.12%	2.45%	3.38%	9.62%	3.73%	10.51%
RTHM	2	10.67%	1.26%	7.88%	0.40%	9.82%	-9.64%	12.16%	-13.18%	13.99%
	4	11.97%	-0.56%	8.45%	1.22%	7.94%	-12.24%	11.12%	-9.76%	12.20%
	8	12.06%	6.74%	5.94%	7.17%	5.49%	-1.48%	14.95%	-0.41%	14.76%

Table 5.10: Experiment 5 correlation between results.

	Correlation	2 PCB	4 PCB	8 PCB
AIM	ELPI-to-GLPI	0.3855	-0.3318	0.1286
	ELPI-to-TLPI	-0.0347	-0.0631	0.3135
PAPM	ELPI-to-GLPI	*0.7645*	*0.9925*	*0.9790*
	ELPI-to-TLPI	*0.7871*	*0.7900*	*0.9441*
RTHM	ELPI-to-GLPI	-0.3413	0.5210	-0.1926
	ELPI-to-TLPI	-0.1542	0.5443	-0.4433

5.5.1 AIM

The data is collected for resolution to the hypothesis that the worst traditional panel design will yield larger component placement times per component than those results obtained through either of the best panelization designs (global or Traditional Best). The global and two-stage approaches are the means by which the panelization results are obtained.

It is seen from Table 5.6 that for the two PCB situation, 90% of both the TLPI and GLPI data show positive gains, for an average of 6% and 3% for TLPI and GLPI, respectively. The confidence interval for both measures, given in Table 5.9, both show a negative lower value, signifying that there is the possibility that including panelization elements in this experiment could result in either no improvements or even a small increase in cycle time for a randomly generated PCB component layout. Though there is one negative result for both the TLPI and GLPI samples, the main cause for the confidence intervals to have a slightly negative lower value is due to the large standard deviations. It should also be noted that these results were obtained from experiments where there is only one panel design and four PCB rotation sets possible for this two PCB panel, AIM scenario, thus presenting few panel design and PCB rotation set alternatives available for a solution. Nevertheless, the confidence intervals are largely positive over most of their ranges, with that for the TLPI showing a more positive overall tendency that for the GLPI.

For the four-PCB panel, Table 5.6 shows that 90% of the TLPI and 100% of the GLPI data show positive gains. Table 5.9 shows higher means with higher standard deviations for both measures. The confidence intervals for both measures are positive over their entire ranges, and nearly the same.

Table 5.6 fails to show any trend of the ELPI in predicting either the TLPI or the GLPI outcomes for any of the panel scenarios, at least with any regular consistency. A simple correlation analysis on the data supports that conclusion for any of the panel scenarios for the AIM, as shown in Table 5.10.

The eight-PCB panel scenario tests show positive TLPI and GLPI for 100% of the tests conducted. The confidence intervals are similar to that of the four-PCB panel results shown in Table 5.9. The correlation results of Table 5.10 show positive results for this situation, but they are so low as to be considered inconclusive.

It appears that for randomly-generated PCB layouts with component distributions centered on or near the PCB geometric center, the panelization approach can consistently show positive improvements for the AIM scenario, as evidenced by the mostly or all positive confidence intervals. However, predicting when a *p*articular PCB and panel design are guaranteed to produce a positive and large TLPI or GLPI is not supported by the ELPI in this particular experiment.

5.5.2 PAPM

The PAPM random PCB layout experiments show the most positive results of the three machine cases. The ELPI, TLPI and GLPI results of the PAPM random PCB layout tests are presented in Table 5.7 for the two, four and eight-PCB panel scenarios. This table shows 100% of the TLPI and the GLPI being positive for all three of the panel scenarios. The confidence intervals given for the GLPI are positive throughout their ranges for the GLPI in the two, four and eight-PCB panel tests. The TLPI confidence intervals are mostly positive,

though the two PCB scenario shows a slight negative value at the lower end; this value is near zero, however.

An initial explanation of the strong, positive results for the PAPM can be given as due to the large number of panel design alternatives available to the problem. In the eight-PCB situation, there are 36 panel pattern alternatives given in Appendix F; this is three times as many as required for the AIM and RTHM situations.

A second explanation can be developed from the nature of the PAPM head movement. On average, the head must travel farther to place a component than must the AIM for a similar panel layout, due to the necessary required and unrequired motions described in section 3.5.1. Thus, very slight differences between any PCB component layouts, relative to their distribution about their geometric centers, will be manifest more in placement solutions for the PAPM than for the AIM. The greater traveling requirements of the PAPM placement head should also result in larger estimator values, thus spreading apart the differences between any given estimator value for one panel design verses another. This tendency is evidenced by the lack of any ELPI below 2% for any of the randomly generated PCB component layouts.

5.5.3 RTHM

The GLPI, TLPI and GLPI results of the RTHM situation for experiment 5 are presented in Table 5.8 for the two, four and eight-PCB panel layouts. The RTHM situation presented the most difficult machine in which to quantify positive results for the random PCB component layout.

The two and four-PCB panel tests showed nearly the same results, with 50% of the TLPI and 60% of the GLPI having negative values. The confidence intervals for these particular tests had strongly negative lower values, with averages of nearly zero. These averages alone indicate that there is little help from panelization elements for the two and

four-PCB scenarios.

The eight-PCB panel showed the most consistent results, with positive TLPI and GLPI for 80% and 90% of the results, respectfully. The confidence intervals of both the TLPI and the GLPI show negative lower values, though both of these are near zero and the upper value is over 14% for both.

The conclusions from the experiment 5 results are that the RTHM problem, which competes the timing of three subsystems in its operation, is only partially benefitted by panelization consideration under the difficult environment of random component location generation. There must be a large selection of panel design alternatives, such as is present for the eight-PCB panel experiments, in order for the panelization approach to consistently improve the LPI in this situation. These results must also be viewed in the context of the machine parameters defined for the problem, as discussed in section 4.3.

5.6 General Observations

This section addresses observations noted across multiple experiments. The experiments under consideration are experiments 2, 3, 4 and 5; experiment 1 was purely for validating a portion of `panelizer` and thus is not a consideration in this section.

Figures 5.42, 5.43, and 5.44 graphically display the TLPI and GLPI for the different experimental instances. Though the AIM and RTHM charts have the same general trends, the AIM LPI is generally the greater of the two on a case-by-case basis.

For case A with two PCB, it is noted that there can be one panel design favored over another, even for PCB layouts that are perfectly centered about the PCB geometric center. The LPI for these situations is slight, however, being under 10% when such a situation is present.

The implementation of panelization as a means by which to reduce the component

placement time can have only limited success if all the alternative panel designs yield similar component placement results. Related to this issue is the fact that, for the two-stage panelization analysis method, the separation of *estimated* panel design ELPI is critical for the designation of good cycle time minimization candidates for the traditional GA approach. In this research effort, one Traditional Best and Traditional Worst panel design are selected from the algorithm of section 3.5. It is not uncommon that there would be several panel designs (both patterns and PCB rotation sets) which are very close to those extreme values. The eight-PCB, PAPM, case C test for experiment 2 was examined in detail by printing all enumerated results ($2^8 \times 36$ patterns $= 9,216$). Even though the worst estimator value (244.319) was nearly twice that of the best value (127.543), there were 3 panel design alternatives within 99.5% of that best value. Based upon the assumption that the panel designs producing the lowest estimator values would produce the best component placement times, multiple panel designs would have nearly the same potential for success.

The most obvious machine type that can take advantage of panelization effects is the PAPM. This may be due to several factors, including

- Travel for the PAPM head involves leaving the panel periphery for every component placement.
- More patterns are available for investigation than for the AIM or RTHM, since the PAPM environment differentiates between the panel orientation on the mounting table.

As the number of PCB per panel increases, the more patterns are available in the panel designs for the component minimization process. However, this does not necessarily lead to the conclusion that more patterns will allow for a correspondingly greater difference (in terms of their component placement times) between the best and worst design results under all conditions. In experiment 2 for the AIM and RTHM machine types, the LPI measures increased from the two-PCB situation to the four-PCB situation, but that value decreased to the eight-PCB panel. This decrease in these measures can be explained to some extent by

figures 5.32 and 5.36. The best design predicted by the estimator algorithm for the eight-PCB panel requires that some long distances be traveled by the placement head (or table fixture), even though the best design will result from an attempt to minimize those distances. In the four-PCB panel situations, these long moves are completely eliminated due to the virtue of the pattern available. Thus, increases in the number of PCB on a given panel may not necessarily correspond to similar increases in the LPI for a given machine type. It should be noted that these longer moves are occurring as a result of the highly idealized, tightly grouped component patterns created for these experiments. More general layouts (such as those for Experiment 5) should not have such extremes between the longest and shortest head moves. This factor may be why the RTHM Experiment 5 experiments for the eight-PCB panel yielded moderately good results, statistically, for randomly generated component locations, yet was so inaccurate for cases A, B and C. For the idealized, tightly grouped

Table 5.11: Distance from PCB geometric center to center of component distribution.(mm)

Case	Two PCB	Four PCB	Eight PCB
A	0	0	0
B	87.32	49.50	42.01
C	149.05	86.27	65.37

component layouts of experiment 2, the difference between the worst possible scenario and the best possible scenario decreases (to some extent) due to the nature of the experimental panel design. The boundary of the panel is the same in all three panel layouts (two, four, and eight-PCB layouts). This fixed panel boundary was used to establish a common placement area for all the experiments. Equally important was the fact that a fixed set of panel dimensions allowed for easier verification of the input files for the 285 experimental computer runs. As a consequence of the fixed panel boundary, as the number of PCB increases to the next PCB-per-panel level, the eccentricity of the component groups for a given case (A, B, or C) will *decrease*, as is shown by Table 5.11. The PAPM shows the decrease most uniformly, where the minimum distances required for placement are driven by the distance from each component to the feeder rack off the side of the PCB. With less difference between

the different possible orientations of the PCB between successive cases, the PAPM has less opportunity to improve for each case. However, since the PAPM requires that the head travel off the panel for each component placement, the net distance improvements are still substantial as the PCB component layout eccentricity decreases.

The AIM and RTHM situations show the LPI measures *increasing* from the two to four-PCB panel experiments, then decreasing for the eight-PCB panel experiments. This hump in the charts is counter to the above explanation due to eccentricity decrease in the PCB designs. However, there are two other factors involved: The machine types and the additional PCB rotation sets produced by square PCB (as apposed to the rectangular PCB for cases A and C). The machine types AIM and RTHM both share a TSP-type element associated with their placement minimization problems. As such, the orientation of the PCB component groups relative to other PCB component distributions is manipulated to reduce the overall placement solutions. This aspect is different from that of the PAPM, where the PCB component group orientations are improved through their manipulation relative to the off-panel feeder slot locations. Thus, it is possible that the square PCB allow for better settling of the component groups than rectangular PCB, due to more degrees of freedom. The AIM and RTHM benefit better from these additional degrees of freedom, since they can orient off two edges of each of the adjacent PCB in the four-PCB panel. The PAPM, on the other hand, is only orienting off the bottom edge in all the cases, due to the feeder slots being located relative to that direction for these particular experiments. The additional degrees of freedom associated with the square PCB may not allow as much benefit for the PAPM. However, it should be noted that the PAPM is showing better LPI results for all the cases when compared to the other machine types, even though the decreasing LPI trend is present through each successive case of increasing PCB component distribution eccentricity.

In section 5.5, it was noted that there was strong correlation between the ELPI and the TLPI/GLPI measures for the PAPM tests, but not for the AIM and RTHM. This observation is noted under the circumstances of random designs, which has been defined as a difficult scenario against which to implement panelization. However, one can observe

Table 5.12: Experiment 2 correlation between results.

	Correlation	2 PCB	4 PCB	8 PCB
AIM	ELPI-to-GLPI	0.9853	0.9961	0.9806
	ELPI-to-TLPI	0.9908	0.9948	0.9045
PAPM	ELPI-to-GLPI	0.9999	1.0000	1.0000
	ELPI-to-TLPI	1.0000	1.0000	1.0000
RTHM	ELPI-to-GLPI	0.9913	0.9999	0.8045
	ELPI-to-TLPI	0.9910	0.9943	0.9180

the close correlation of the ELPI with the other two measures for *any* machine type in the experiment 2 data. Table 5.12 shows the correlation between the ELPI against both the TLPI and GLPI, using the three cases (A, B, C) as the basis for each data point. This information is presented with the realization that correlation with only three data points may not give conclusive results. However, the correlation results are strong, being over 0.9 for all but one of the entries in Table 5.12, and that entry (RTHM with eight PCB) is still a respectable 0.8045.

The ELPI may be correlated to the two LPI measures under various circumstances, but it is difficult to predict the actual magnitude of a TLPI or GLPI result based upon a single ELPI result. This problem is due to the estimator function formulation, which is based upon component group distances. The formulation thus does not directly attempt to measure an optimal result, and therefore yields magnitudes which do not relate to the final, GA result from the component placement algorithm. This observation is especially true for the RTHM; the estimator gives results in terms of distance (millimeters), but the RTHM solutions are in units of time (seconds).

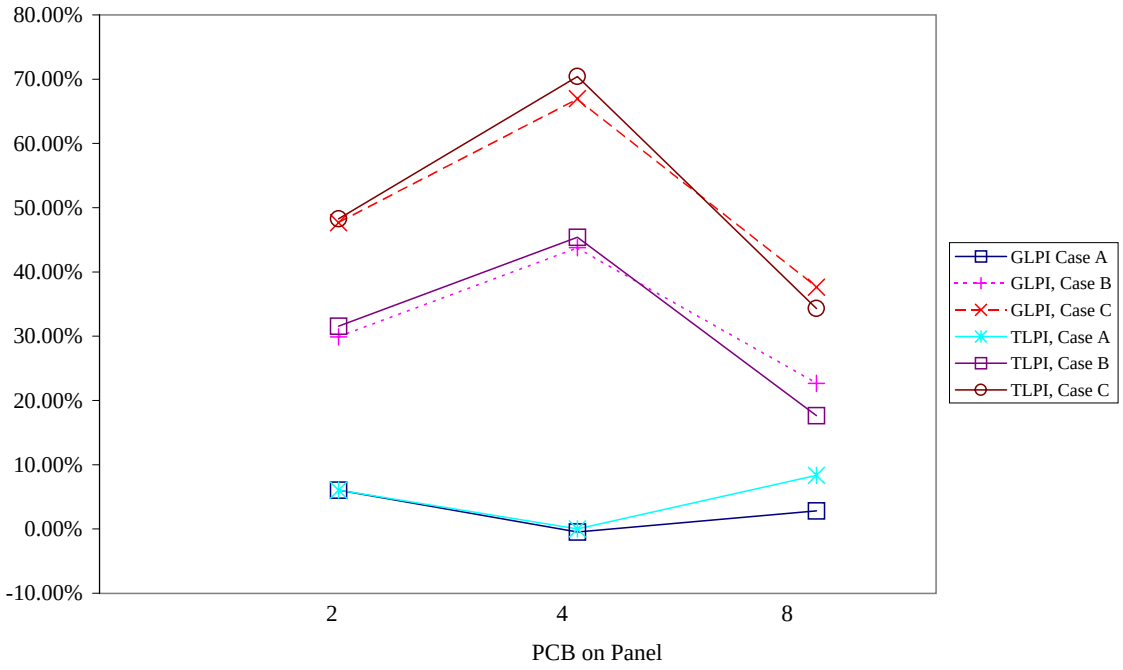


Figure 5.42: Comparison of AIM experiment 2 results over PCB population.

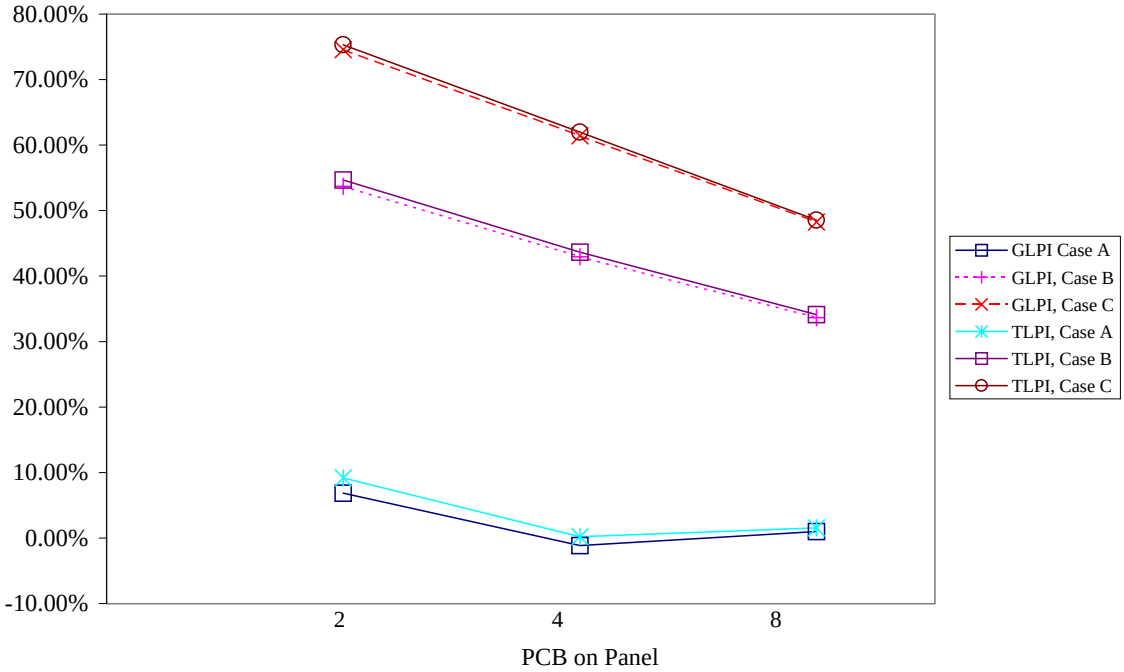


Figure 5.43: Comparison of PAPM experiment 2 results over PCB population.

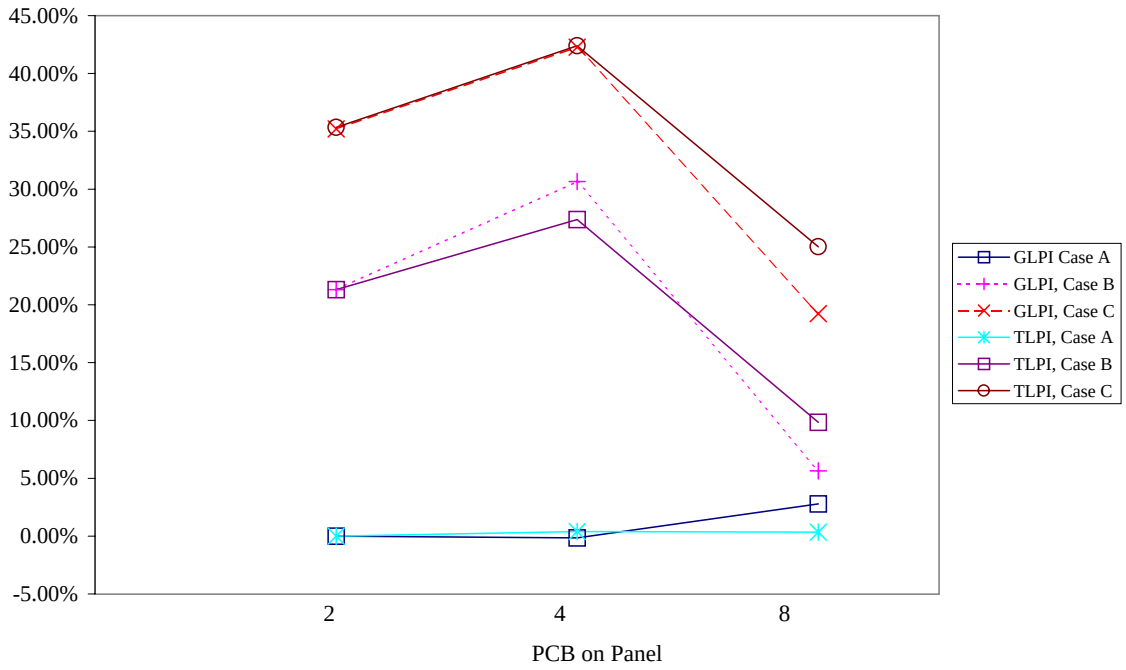


Figure 5.44: Comparison of RTHM experiment 2 results over PCB population.

Chapter 6

Conclusions and Future Research

This research has introduced the concept of panelization into the manufacturing research genre as a means by which to reduce the component placement time of printed circuit panels. The results of this research are presented in order to add new knowledge to both the electronic manufacturing and design-for-manufacturing research areas. This chapter will describe some of the contributions from the research presented in this document. Also, future research opportunities will be identified.

6.1 Summary

The approach behind minimizing the component placement time was begun by developing mathematical definitions of the panel design with the goal of incorporating those terms in a minimization heuristic. The heuristic selected was the GA; this method traditionally included only the component placement sequence and component type allocation to feeder slots (when needed) as part of its independent variable set. The heuristic was redefined in order to include two additional panel design variables: 1) The panel pattern alternative and 2) the PCB rotation set. The GA heuristic was developed to progress through the problem

space to a solution while simultaneously changing four variables. This unifying heuristic application was called the global approach.

A second heuristic was later developed and named the two-stage approach. This approach selected a "best" panel design as a result of applying and ranking all possible panel designs via estimator function scores. These estimations produced scores for each design, the values of which were based upon approximate location assumptions associated with the geometric distances of component groupings relative to each other. In one machine type (PAPM), the distances were calculated as those between the component groups and the feeder slot locations. Once the panel design was established by this method, the second stage of the heuristic was the application of the GA on the now-traditional problem of selecting the component placement sequence and the component allocation (as needed). The goal of the second stage is to produce low (though not necessarily optimal) component placement times.

6.1.1 Objectives addressed

The question associated with the research hypothesis (section 1.3.2) was whether or not the application of panelization to the component placement time minimization problem could further reduce placement times compared to when panelization was not considered. The answer to this question is conditioned by both the problem definition as well as the experimental results found in this research.

The first part of the answer to the question is found in the definition of the problem itself. The traditional analysis was assumed to have a single panel design selected by some means which may or may not have component placement time reduction as a consideration. The assumption is that this objective is not typically a factor for panel design, since little mention of panelization has been found in literature. With this assumption valid, the improvement of component placement times would be possible only if the traditional selection method yielded a design different than that of the global or two-stage approach. Thus, the

largest possible improvement in component placement time could only occur if the worst design were selected in the traditional selection process and then compared against the best design selected from either the global or two-stage approaches. This concept is the basis for both the GLPI and TLPI formulations in section 4.3. Clearly, the component placement improvements could be a value less than the GLPI or TLPI. This even would occur if the traditional panel design selection method (by chance or by design) determined a panel design with similar cycle time results as that panel design determined through the global or two-stage approaches. If the selected design for the traditional analysis were *the same* as those found through the two new approaches, then any differences between these results would be due solely to the characteristic associated with GA heuristics in producing near but not identical solutions for the same problem. In such a situation, the GLPI and TLPI should be essentially zero.

The second answer to the hypothesis is based upon the experimental results. In some problem scenarios, the incorporation of panelization had too little or no impact on the component placement time reduction problem when compared to the traditional method. If the magnitudes of the LPI were small or even zero in some problem scenarios, then there would be no chance for the implementation of panelization in the component placement problem to be a factor in the solutions. Experiment 3 and case A scenarios of experiment 2 show that uniformly distributed component locations on the PCB will yield near-zero GLPI and TLPI if that arrangement is centered on the PCB geometric center. The near-zero LPI results are the expected numerical results from scenarios where panelization has no impact in the component placement time problem.

The development and analysis of experiments for different machine types, PCB component layouts and panel design scenarios were a result of addressing the research objective (section 1.3.2). This objective was to show cycle time improvements through the consideration of panelization. As a part of this research, there were *two* approaches developed which produced very similar results in the experiments. As discussed above, the improvements were assumed to be the LPI measures developed; the GLPI was developed as the improvement

measure for the global approach and the TLPI for the two-stage approach. The experimental scenarios affected the magnitude of the outcomes; thus, the effect of panelization on improving component placement time results should be addressed in terms of the experimental scenarios under consideration.

6.1.2 Results

The findings of this research are addressed in terms of the GLPI and TLPI. As discussed in section 6.1, these results are the largest *potential* benefits obtained by including panelization.

The most general results can be stated in neutral terms: If the estimator difference, ELPI, results in a low value close to zero, then the implementation of panelization in the component placement time reduction calculations will not reduce cycle times. In the context of the experiments conducted as part of this research, that low ELPI threshold is approximately 4%.

The PAPM shows the most consistent, positive potential cycle time reductions in all the experimental scenarios. When the ELPI is above the minimum 4%, the GLPI and TLPI ranges from approximately 4% to 75%. The more specific findings are addressed in terms of the experiments, for those results having ELPI above 4%:

- *Experiment 2:* GLPI and TLPI range from 10% to 75% for all machine types and all panel design scenarios. The ELPI track closely to the GLPI and TLPI for all situations (Table 5.12).

- *Experiment 3:* The ELPI were below 4%; the GLPI and TLPI results also were below 4%, supporting the value of ELPI predicting low results for this situation with the PAPM and RTHM.

- *Experiment 4:* The PAPM GLPI and TLPI were both about 6%, exceeding the prediction of 2.6% by the ELPI. For the RTHM scenario, the GLPI and TLPI were 1.7%

and 4%, respectively. The ELPI was 12%.

- *Experiment 5:* AIM and PAPM TLPI averaged 4% to 14%; the average GLPI was 3% to 12.5% for all panel design scenarios. The RTHM average for both measures was essentially zero for the two and four-PCB situations. For the eight-PCB RTHM situation, the GLPI and TLPI averages were both 8%.

Generally positive panelization results (either LPI above 4% or an ELPI prediction of low results) are noted for all experiments except experiment 4. This experiment addresses an industrial PCB design (illustrated in Figure 4.10). The results for that experiment, particularly for the RTHM scenario, show only small improvements which were not predicted well by the ELPI. This PCB design has a high number of components for the number of components on the PCB. The design also has a concentration of its component locations along one axis. Further study of this type of situation may be in order.

6.2 Contributions

The most important contribution from this research is the introduction of panelization as a means by which to reduce the component placement time of printed circuit panels. Such a concept is not commonly found in the electronic assembly manufacturing literature surveyed as part of this research.

Part of this contribution is the panelization definition in itself. The basic elements of panelization were identified in terms of the panel design:

- The pattern alternatives formed by the individual PCB boundaries.
- The individual rotations allowed for the PCB within the pattern alternatives. This element was termed the PCB rotation set.

These two definitions allowed the modification of the component placement cycle time reduction heuristic to include panelization in the problem.

Another contribution was the creation of the estimator functions for the particular machine types. These functions were used as the first part of the two-stage method discussed above. They were also used to find an estimated worst panel design for comparison in both the GLPI and TLPI measures. Another useful purpose of the estimator method was in using the estimator approximate scores themselves as a predictor (the ELPI) of the usefulness in applying the panelization to a given experimental scenario. This predictor method was to some extent dependent on the experimental scenario under consideration.

6.2.1 General Rules for Application

Based upon the experimental results of this dissertation, some general rules for application of panelization to a problem can be extrapolated for limited situations:

- *When to investigate panelization for AIM and PAPM.* A minimum ELPI which will guarantee a positive LPI is not supported by the experiments conducted. However, it can be noted from these experiments that positive LPI was present for ELPI above 4% for AIM and PAPM situations, for two, four and eight-PCB panels. Thus, panelization is a reasonable undertaking for most scenarios with these two machine types; the only exception being a very uniform distribution of the component locations.

- *When to investigate panelization for RTHM.* For highly grouped, offset PCB designs (such as in experiment 2), looking into panelization can prove beneficial if the ELPI is above 1%. However, if the component distribution is more dispersed and difficult to quantify, then many PCB should be present in the panel (in these experiments, eight PCB) in order create more panel design alternatives to potentially improve the solution. In such a situation, the ELPI should be above 14%, based upon results in experiment 5.

6.3 Future Research Opportunities

Further research can use the basic research presented here to produce tangible benefits when considering new types of PCB circuitry layouts, various assembly machine types and PCCA production practices.

6.3.1 Characterization of PCB component layouts

This research roughly characterized a generic PCB design in terms of number of components and component types, the boundary dimensions, and the component locations distribution. Each of these parameters were established loosely, based either upon a literature survey or via geometry assumptions. The experimental results in this research show that the potential for component placement time reduction can be large when panelization is considered, but the PCB layout characteristics under study can make a considerable difference in the magnitude of the result.

There was little research found in the study of PCB component layout and its direct impact on component placement times. A regimented characterization of PCB component layouts would help in developing experiments for future design-for-manufacturing research which specifically addresses component placement time reduction or minimization.

6.3.2 Expansion into Entire Assembly Line

The techniques developed in this research are a first step in the integration of the panel design elements with those of assembling those panels in the manufacturing environment. As such, the definition of the problem is idealistic, assuming that a particular panel will be manufactured on a single machine type. In reality, there are often several machines in a manufacturing line which will operate on the same panel; these machines, in series, will ultimately form the final printed circuit products from the panel. This aspect of industrial

practice was presented in section 1.3.2. These machines in the manufacturing line can be of the same or different types. The GA technique developed in this research can be modified to address this situation, such that the best panel for that assembly line can be selected in order to reduce the cycle time of the panel for that line.

6.3.3 Specific Machine Types

The machine types examined in this research were idealistic machines, based upon surveys of research literature. Examples of the assumptions and parameters upon which the machine operations were based are listed in section 1.2.2 and include constant placement head velocities, uniform speeds for different component types, and so on. Future panelization research can be conducted on these machines with a more detailed and varied set of machine parameters. As an example, problems can be addressed which account for the variable speeds of the head in operation, since the head ramps up to its maximum velocity only when it travels a sufficient distance. The same speed variability applies to moving feeder banks for RTHM-type machines as well; the feeder can move at a faster average velocity as it travels across several feeder slots instead of only traveling from one adjacent slot to the next.

In the PCCA industry, there are a variety of machine types which deviate from the three discussed in this research. These machines are conceptually similar to the AIM, PAPM or RTHM, but differ in their subsystems to some degree. Section 2.3 presents many other machine types upon which research has already been investigated; panelization can be addressed for these machine types or for newer machines currently in industry.

The GA technique was used in this research to analyze multiple machine types, but when only one machine is to be considered, algorithms tailored for improving solution times for that machine can be incorporated into the panelization research. The two-stage technique, developed as a means by which to estimate best and worst panel designs, can be modified to accept the machine-specific algorithms. The estimated best panel design (or even a set of potentially best designs) can be given to the machine algorithm in order to produce a low

cycle time which takes advantage of panelization elements.

The global analysis technique for panelization would be suitable for assembly machine manufacturers to incorporate into their machine optimization software. In current practice, the machine must be programmed for the PCB layouts and/or the panel design. Research into panelization for specific machine types would result in the placement machine having the option to allow for optimization of the panel design along with the component placement and allocation solutions.

The global approach is possible for use in electronic assembly facilities as well, but it would require extensive, custom programming by the designer or manufacturing engineer and thus may be difficult to implement in practice for the PCCA facility. Therefore, the two-stage technique using the estimator functions would be a good alternative for these applications. The genetic algorithm need not even be used for the second part of the two-stage approach (after the potentially best designs have been selected), if the user has another traditional technique available specifically developed for the machine type under consideration.

Bibliography

[1] Javad Ahmadi, Stephen Grotzinger, and Dennis Johnson. Component allocation and partitioning for a dual delivery placement machine. *Operations Research*, 36(2):176–191, March-April 1988.

[2] Reza H. Ahmadi and Panagiotis Kouvelis. Staging problem of a dual delivery pick-and-place machine in printed circuit card assembly. *Operations Research*, 42(1):81–91, January-February 1994.

[3] Michael O. Ball and Michael J. Magazine. Sequencing of insertions in printed circuit board assembly. *Operations Research*, 36(2):192–201, March-April 1988.

[4] J. E. Beasley. An algorithm for the two-dimensional assortment problem. *European Journal of Operational Research*, 19:253–261, 1985.

[5] J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64, Jan-Feb 1985.

[6] K. Broad, A. Mason, M. Rönnqvist, and M. Frater. Optimal robotic component placement. *Journal of the Operational Research Society*, 47:1343–1354, 1996.

[7] R. Bruns. Direct Chromosome Representation and Advanced Geneitc Operators for Production Scheduling. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 352–359, San Mateo, CA, 1993. Morgan Kaufmann.

[8] M. L. Chambers and R. G. Dyson. The cutting stock problem in the flat glass industry-selection of stock sizes. *Operational Research Quarterly*, 24(4):949–957, 1976.

[9] D. Chan and D. Mercier. IC insertion: An application of the travelling salesman problem. *International Journal of Production Research*, 27(18):1837–1841, 1989.

[10] Chi-Ming Chang and Li Young. A simultaneous-mounting process for automated printed circuit board assembly. *International Journal of Production Research*, 28(11):2051–2064, 1990.

[11] F. Chauny and R. Loulou. Lp-based method for the multi-sheet cutting stock problem. *INFOR*, 32(4):253–264, 1994.

[12] Nicos Christofides and Charles Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25(1):30–44, 1977.

[13] Frank Classon. Step-by-step SMT. 5. Component placement. *Surface Mount Technology*, spec. suppl:26, 28–9, 1995.

[14] Jr. Coffman, E. G., and P. W. Shor. Average-case analysis of cutting and packing in two dimensions. *European Journal of Operational Research*, 44:134–144, 1990.

[15] R. de Souza and Lijun Wu. CPS: A productivity tool for component placement in multi-head concurrent operation PCBA machines. *Journal of Electronics Manufacturing*, 4(4):71–79, 1994.

[16] R. de Souza and Lijun Wu. Intelligent optimization of component onsertion in multi-head concurrent operation PCBA machines. *Journal of Intelligent Manufacturing*, 6(4):235–43, August 1995.

[17] A. Dikos, T. M. Tirpak P. C. Nelson, and Weihsin Wang. Optimization of high-mix printed circuit card assembly using genetic algorithms. *Annals of Operations Research*, 75:303–24, 1997.

[18] Alan A Farley. Selection of stockplate characteristics and cutting style for two dimensional cutting stock situations. *European Journal of Operational Research*, 44:239–246, 1990.

[19] T. C. Fogarty. Varying the Probability of Mutation in Genetic Algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109. Morgan Kaufmann, 1989.

[20] D. B. Fogel. *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*. IEEE Press, 1995.

[21] Douglas D. Gemmill. Solution to the assortment problem via the genetic alogorithm. *Mathematical Computer Modeling*, 16(1):89–94, 1992.

[22] Douglas D Gemmill and Jerry L. Sanders. Approximate solutions for the cutting stock 'portfolio' problem. *European Journal of Operational Research*, 44:167–174, 1990.

[23] P. C. Glimore and R. E. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations Research*, 13:94–120, 1965.

[24] David E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[25] John J. Grefenstette. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[26] Stephen Grotzinger and Anna Sciomachen. A petri net characterization of a high speed placement machine. In *1988 Proceedings of the 38th Electronics Components Conference*, p. x+664, pages 64–68, Los Angeles, CA, USA, May 1988. IEEE; Electron. Ind. Assoc, IEEE; New York, NY, USA.

[27] H. O. Günther, M. Gronalt, and R. Zeller. Job sequencing and component set-up on a surface mount palcement machine. *Production Planning and Control*, 9(2):201–211, 1998.

[28] Eleni Hadjiconstantinou and Nicos Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

[29] Robert W. Haessler and Paul E. Sweeney. Cutting stock problems and solution procedures. *European Journal of Operational Research*, 54:141–150, 1991.

[30] H. C. Herz. A recursive computing procedure for two-dimensional stock cutting. *IBM Journal of Research and Development*, 16:462–429, 1972.

[31] A. I. Hinxman. The trim-loss and assortment problems: A survey. *European Journal of Operational Research*, 5:8–18, 1980.

[32] J. Holland. *Adaptation in Natural and Artifical Systems.* The University of Michigan Press, Ann Arbor, 1975.

[33] A. Homaifari, S. H. Y. Lai, and X. Qi. Constrained Optimization via Genetic Algorithms. *Simulation*, 62(4):242–254, 1994.

[34] C. R. Houck, J. A. Joines, and M. G. Kay. A Genetic Algorithm for Function Optimization: A Matlab Implementation. NCSU-IE Tech Report 95-09, North Carolina State University, 1995.

[35] Yu-Wen Huang, K. Srihari, Jim Adriance, and George Westby. Heuristic based multiple batch placement sequence optimization. In *Proceedings of the Technical Program. NEPCON West'93*, volume 3, pages 1598–607, Anaheim, CA, USA, February 1993. Proceedings of NEPCON West, Reed Exhibition Companies; Des Plaines, IL, USA.

[36] Yu-Wen Huang, K. Srihari, Jim Adriance, and George Westby. A solution methodology for the multiple batch surface mount pcb placement sequence problem. *Transactions of the ASME*, 116:282–289, December 1994.

[37] P. Husbands, F. Mill, and S. Warrington. Genetic Algorithms, Production Plan Optimization, and Scheduling. In H. P. Schwefel and R. Männer, editors, *Proceedings of the First International Conference on Parallel Problem Solving from Nature (PPSN)*, volume 496 of *Lecture Notes in Computer Science*, pages 80–84. Springer-Verlag, 1991.

[38] H. S. Ismail and K. K. B. Hon. The nesting of two-dimensional shapes using genetic algorithms. In *Proceedings of the Institution of Mechanical Engineers, Part B (Journal of Engineering Manufacture)*, volume 209 of *B2*, pages 115–124, 1995.

[39] Douglas C. Jeffery. Designing PC cards. *Printed Circuit Fabrication*, 18(8):24–26, August 1996.

[40] Zhiming Ji, Ming C. Leu, and Hermean Wong. Development and implementation of linear assignment algorithm for assembly of pcb components. In Editor, editor, *Proceedings of the 1993 ASME International Electronics Packaging Conference*, number 1 in 4, pages 365–371, Binghamton, NY, USA, September 1993. ASME EEP, ASME, NEW YORK, NY, (USA).

[41] Andy Kowalewski. Designing PWBs for cost-effective manufacturing. *Australian Electronics Engineering*, 24(7):32,34,36,38, 1991.

[42] Ratnesh Kumar and Haomin Li. Integer programming approach to printed circuit board assembly time optimization. *IEEE Transactions on Coponents, Packaging, and Manufacturing Technology–Part B*, 18(4):720–727, November 1995.

[43] Thomas L. Landers, William D. Brown, Earnest W. Fant, Eric M. Malstrom, and Neil M. Schmitt. *Electronics Manufacturing Processes*. Englewood Cliffs, NJ, 1994.

[44] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Inc., New York, NY, 1991.

[45] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem*. John Wiley and Sons Ltd., New York, NY, 1985.

[46] Timo Leipälä and Olli Nevainen. Optimization of the moverments of a component placement machine. *European Journal of Operational Reserach*, 38:167–177, 1989.

[47] V. Jorge Leon and Brett A. Peters. Replanning and analysis of partial setup strategies in printed circuit board assembly systems. *International Journal of Flexible Manufacturing Systems*, 8(4):389–411, October 1996.

[48] M. C. Leu, H. Wong, and Z. Ji. Planning of component placement/insertion sequence and feeder setup in pcb assembly using genetic algorithm. *Transactions of the ASME*, 115:424–432, December 1993.

[49] Ming C. Leu, Hermean Wong, and Zhiming Ji. Genetic algorithm for solving printed circuit board assembly planning problems. In *Proceedings 1992 Japan-U.S.A. Symposium on Flexible Automation - A Pacific Rim Conference*, volume 2, pages 1579–1586, San Francisco, CA, July 1992. ASME: Inst. Syst., Control and Inf. Eng., Inst. Syst. Control and Inf. Eng; Kyoto, Japan.

[50] Huan-Chung Lin, Pius J. Egbelu, and Chung-Te Wu. A two-robot printed circuit board assembly system. *International Journal of Computer Integrated Manufacturing*, 8(1):21–31, 1995.

[51] R. J. Linn. Component placement error model for surface mounted technology. In B. Bidanda R. G. Askin and S. Jagdale, editors, *Proceedings of the Fifth Industrial Engineering Research Conference*, number 2 in p. xix+824, pages 91–6, Minneapolis, MN, USA1, May 1996. ASME: Inst. Syst., Control and Inf. Eng., Inst. Ind. Eng; Norcross, GA, USA.

[52] L. F. McGinnis, J. C. Carlyle, L. Cranmer, G. W. DePuy, K. P. Ellis, and C. A. Tovey. Automated process planning for cirucuit card assembly. *IIE Transactions*, 24(4):18–30, 1992.

[53] L. F. McGinnis, J. C. Carlyle, L. Cranmer, G. W. DePuy, K. P. Ellis, and C. A. Tovey. Circuit card assembly process planning. In B. Bidanda R. G. Askin and S. Jagdale, editors, *Proceedings of the ASME Winter Annual Meeting*, pages 91–6, New Orleans, LA, USA, May 1993. ASME.

[54] J. McLenaghan. PCMCIAs: Mounting multiple technologies. *Surface Mount Technology*, 9(7):36–9, July 1995.

[55] Z. Michalewicz. *Genetic Alogrithms Plus Data Structures Equals Evolution Programs.* Springer-Verlag, New York, New York, third, revised and extended edition, 1996.

[56] Z. Michalewicz, J. D. Schafier, H.-P., D. B. Fogel, and H. Kitano, editors. *Proceedings of the First IEEE International Conference on Evolutionary Computation*, Orlando, June 1994. IEEE Press.

[57] Harvey Miller. Top 15 PCB makers. *Printed Circuit Fabrication*, 19(12):40–43, December 1996.

[58] Joe Moran and James Maddox. Scoring tricks. *Printed Circuit Fabrication*, 20(1):40–41, January 1997.

[59] Vedran Mornar and Behrokh Khoshnevis. A cutting stock procedure for printed circuit board production. *Computers in Industrial Engineering*, 32(1):57–66, 1997.

[60] L. K. Moyer and S. M. Gupta. Simultaneous component sequencing and feeder assignment for high speed chip shooter machines. *Journal of Electronics Manufacturing*, 6(4):271–305, December 1996.

[61] L. K. Moyer and S. M. Gupta. SMT feeder slot assignment for predetermined component placement paths. *Journal of Electronics Manufacturing*, 6(3):173–92, September 1996.

[62] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In J.J.Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230. Lawrence Erlbaum Associates, 1987.

[63] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Englwood Cliffs, NJ, 1982.

[64] D. Powell and M. M. Skolnick. Using Genetic Algorithms in Engineering Design Optimization with Non-linear Constraints. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 424–430. Morgan Kaufmann, 1993.

[65] Ray Prasad. Step-by-step SMT. 1. Design for smt. *Surface Mount Technology*, spec. suppl:6–9, 1995.

[66] Ronal J. Rapczynski. Improving manufacturability with good smt design practice. *Surface Mount Technology*, pages 26–29, August 1989.

[67] Subhash C. Sarin. Two-dimensional stock cutting problems and solution methodologies. *Journal of Engineering for Industry, Transactions of the ASME*, 105:155–160, August 1983.

[68] Richard Schneider. Board fabrication, panelization, supplier coordination. In *Proceedings of the Technical Program. National Electronic Packaging and Production Conference (NEPCON)*, volume 3, pages 2209–2211, Chicago, IL, USA, February 1991. Proceedings of NEPCON West, Reed Exhibition Companies; Des Plaines, IL, USA.

[69] W. R. Shih, K. Srihari, and J. Adriance. Expert system based placement sequence identification for surface mount pcb assembly. *The International Journal of Advanced Manufacturing Technology*, 11(6):413–424, 1996.

[70] Vern Solberg. Containing assembly costs through design for manufacturability. *Electronic Packaging and Production*, 33:36–22,36–25, 1993.

[71] Yi-Chen Su, Collin Wang, Pius J. Egbelu, and David J. Cannon. A dynamic point specification approach to sequencing robot moves for pcb assembly. *International Journal of Computer Integrated Manufacturing*, 8:303–324, 1995.

[72] Paul E. Sweeney and Elizabeth R. Patternoster. Cutting and packing problems: A categorized, application-orientated research bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[73] G. Syswerda. Adapting Operator Probabilities in Genetic Algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69. Morgan Kaufmann, 1989.

[74] G. Syswerda. Uniform Crossover in Genetic Algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.

[75] G. Syswerda. Schedule Optimization Using Genetic Algorithms. In L. Davis, editor, *Handbook of Genetic algorithms*, pages 332–349, New York, 1991. Van Nostrand Reinhold.

[76] John T. Tester. The need for optimized panel design. In H. Migliore, S. Randhawa, W. G. Sullivan, and M. M. Ahmad, editor, *Proceedings of the Flexible Automation and Intelligent Manufacturing Conference; FAIM98*, pages 365–376, Portland, OR, July 1998. FAIM, Begell House, Inc.

[77] John T. Tester. Reducing electronic panel assembly time via panel design selection. *Journal of Flexible Automation and Intelligent Manufacturing*, 3, In Press, 1999.

[78] L. van de Vall. Optimizing medium-volume placement machine output. *Surface Mount Technology*, suppl. issue(4):4, 6–7, 1997.

[79] P. Y. Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Operations Research*, 31(3):573–586, May-June 1983.

[80] D. Whitley. Using Reproductive Evaluation to Improve Genetic Search and Heuristic Discovery. In J.J.Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 103–115. Lawrence Erlbaum Associates, 1987.

[81] D. Whitley. The GENITOR Algorithm and Selection Pressure: Why Rank-based Allocation of Reproductive Trials is Best. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.

[82] Wilbert E. Wilhelm and John Fowler. Research directions in electronics manufacturing. *IIE Transactions*, 24:6–17, September 1992.

[83] H. Wong and M. C. Leu. Adaptive genetic algorithm for optimal printed circuit board assembly planning. *CIRP Annals*, 42(1):17–20, 1993.

[84] S.H. Yeo and K. H. Yong. A frame-based approach for feeder arrangement and component sequencing in pcb assembly. In Editor, editor, *Proceedings 1994 Japan-U.S.A. Symposium on Flexible Automation - A Pacific Rim Conference*, number 2 in vol. xxx+1564, pages 591–594, Kobe, Japan;, July 1994. ASME: Inst. Syst., Control and Inf. Eng., Inst. Syst. Control and Inf. Eng; Kyoto, Japan.

Appendix A

panelizer.C

A.1 Command arguments for panelizer.C

The program `panelizer.C` is a C++ program which can be run with command line options. This technique is not user-friendly, but allows for batch submission of experiments on the same computer, which can be convenient when conducting multiple experiments.

The command arguments and definitions are given below. All require an integer value after their inclusion in the command line:

- `popsize` The GA population size
- `nconv` The convergence factor for the GA. This is essentially a slope; it is the ratio of the current best score verses that of the BINSRESET GA generations earlier best score. The BINSRESET value is set in the header file, `MYHEADER.h`.
- `ngen` The *maximum* number of generations to run until termination. May be preempted by convergence through the `nconv` termination.
- `prob0` The probability of doing a GA crossover for the pattern list within the composite genome.
- `prob1` The probability of doing a GA crossover for the PCB rotation set list within the composite genome.
- `prob2` The probability of doing a GA OX crossover for the component placement sequence and/or component allocation list within the composite genome.

- `prob3` The probability of doing a GA inversion crossover for the component placement sequence and/or component allocation list within the composite genome.
- `prob4` The probability of doing a GA rotation crossover for the component placement sequence and/or component allocation list within the composite genome.
- `prob5` The probability of doing a GA mutation for any list within the composite genome.
- `aim` Set to 1 for the AIM experiments
- `papm` Set to 1 for the PAPM experiments
- `rthm` Set to 1 for the RTHM experiments
- `seed` A random number seed value to set the random component locations generations engine (for Case F experiments) as well as for GA calculations.
- `repeats` A particular experiment repeated `repeats` times for new GA solutions, but with new seed values each time. The seed is based upon the original `seed` value passed from the command line.
- `replc` Used for batch processing of multiple experiments with different pattern sets and different PCB component layouts. Used in conjuction with `innum`. If component layouts are read in from a file, then each component layout is read in via a `dA#.txt` or `dP#.txt` file. The `dA#.txt` file is for the AIM case; the `dP#.txt` file is for both the PAPM and RTHM cases, which require comoponent types. The # is a file differentiator, so that one can use sequential filenames in a set of program runs for a single process. For example, a `replc 3` will process experiments for `dP1.txt`, `dP2.txt` and `dP3.txt` in sequence, if `innum 1` is designated. It also sequentially processes the required `pattV?#.txt` files, where the "?" represents the number of PCB in the panel.
- `innum` Used for batch processing of multiple experiments with different pattern sets and different PCB component layouts. Used in conjuction with `replc`. The integer value is appended to the `dA` or `dP` in order to form the appropriate, sequential names for the component locations and pattern input files.
- `trad` 1 designates a traditional analysis; 0 signifies a global (panelization) analysis.
- `locgen` 1 designates that random PCB component locations be generated; 0 indicates that these values are read from an input file (typically `dA#.txt` or `dP#.txt`).
- `rect` 1 signals the AutoCad script be written to have rectangles represent component locations; 0 indicates that the component locations be represented by their component identification number, corresponding to the value in the GA solution of "list" 2.
- `pcbb` Number of PCB in a panel.
- `convg` 0 turns off slope convergence termination established by `nconv`; 1 allows slope convergence. Used for the verification cases, where a fixed number of generations must be accomplished.

- `alike` 1 designates that the PCB are all identical; 0 indicates that they are all different. Only the 0 option was used for this research; the 1 option was not fully debugged.
- `pconv` Used for convergence of a proportion of the GA population within a narrow range. A GAlib argument which was not used in the experiments of this research.
- `mprob` Controls the tournament selection probability. Default is 1.0 for the experiments.
- `dumpout` A flag used for debug printing. Not used in experimental runs.

A.1.1 Batch submission examples

Below are three examples of how `panelizer` has been used in this research, with some of the arguments explained:

- **Example 1.** Global analysis, 3 repetitions under same PCB design, 5 replications of the 3 repetitions with different designs, start with dP/A1.txt file (and hence no random PCB component location generation), machine type AIM. The global analysis is keyed by the `trad 0` arguments, and the dA1.txt file is keyed from the `innum 1` arguments. There must be five dA#.txt files, in sequence (i.e., dA1.txt, dA2.txt, . . . , dA5.txt) in order for all five replications to be accomplished. Since the dA#.txt files are defined, the aimpcbV2.txt file must be available; the "2" in that filename is set by the `pcbb 2` arguments. Note that `prob0 0` defines a starting value of 0% for the pattern crossover probability. This value is appropriate for the two PCB panel in the AIM machine, since only one pattern alternative is available for the experiments.

  ```
  panelizer ngen 4000 popsize 100 seed 2 prob0 0 prob1 20 prob2 20 prob3 20
  prob4 20 prob5 10 innum 1 trad 0 replc 5 repeats 3 aim 1 pcbb 2 locgen 0
  rect 1
  ```

- **Example 2.** Traditional analysis, start with dP5.txt (no random PCB component location generation due to `logen 0`, machine type PAPM, eight PCB. The traditional analysis is keyed by `trad 1`. Also, the component identification numbers are printed out for each Autocad script file due to the `rect 0` arguments.

```
panelizer ngen 4000 popsize 100 seed 1 prob0 20 prob1 20 prob2 20 prob3 20
prob4 10 prob5 10 innum 5 trad 1 replc 5 repeats 3 papm 1 pcbb 8 locgen 0
rect 0
```

- **Example 2.** Global analysis, random PCB component/type layout generation due to `locgen 1`, machine type RTHM. `innum 0` is also required for the random PCB layout generation for `panelizer`, but is actually zero by default; the arguments are shown here for illustration. The random PCB component generation results in dP#.txt files not required for the program to run; these files would ordinarily provide such locations and types, but the program generates them internally instead. `locgen 1` also requires a different PCB dimensions data file, `RTHMtest8.txt`, instead of `RTHMpcbV8.txt`, since several bounds are required for the random generation of the number of components, the component types, and the means by which to generate the feeder slot locations based upon the number of component types. All slots are located at a fixed position below the panel and equally spaced apart, but there must be the same number of slots as the number of component types generated for an experiment. The spacing of the slots and their location relative to the panel is specified in `RTHMpcbV8.txt`. Note that three GA will be run on a single PCB layout generated by the program due to `repeats 3` and five PCB layouts will be generated due to `replc 5`. Thus, a total of 15 GA runs will be completed.

  ```
  panelizer ngen 4000 popsize 100 seed 1 prob0 20 prob1 20 prob2 20 prob3 20
  prob4 10 prob5 10 innum 0 trad 0 replc 5 repeats 3 rthm 1 pcbb 8 locgen 1
  rect 0
  ```

A.2 Acknowledgments and copyrights

`panelizer.C` uses objects (or has heavily modified objects) from the GAlib source code developed by Matthew Wall.[1] GAlib code is not in the public domain, but is available at

[1] Copyright (c) 1995-1996 Massachusetts Institute of Technology

no cost for non-profit purposes, and permission is allowed for modification of the code for such purposes. Since the original GAlib source code is quite lengthly and is freely available from the internet, [2] it will not be published in this dissertation. The program created to *use* the GAlib objects, `panelizer`, is 3896 lines of C++ code. The custom header file for `panelizer` is also provided in this appendix.

[2] http://lancet.mit.edu/galib-2.4/

panelizer.C

```
/* ------------------------------------------------------------------------
panelizer.b.
John T. Tester
Based upon locator.b.50
June 30, 1999

This program was written as a result of research conducted at Virginia
  Polytechnic University.  This program uses objects (or has heavily
  modified objects) from the GAlib source code developed by Matthew
  Wall. Copyright (c) 1995-1996 Massachusetts Institute of Technology
  GAlib code is not in the public domain, but is available at no cost
  for non-profit purposes, and permission is allowed for modification
  of the code for such purposes.  GAlib source code is available at
  http://lancet.mit.edu/galib-2.4/
  ------------------------------------------------------------------------ */
#include <string.h>
#include <math.h>
#include "MYHEADER.h"
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <ga/ga.h>
#include <ga/GASStateGA.h>


//  A handy little function, SWAP2 is used in OXover.
//  It's also used in Ordered Crossover, so I use the name
//  SWAP2 to avoid conflicts.
#define SWAP2(a,b) {unsigned int tmp=a; a=b; b=tmp;}

int GAListIsHole2(const GAListGenome<int> &, const GAListGenome<int> &,
            int, int, int);
void rectangle(const GAGenome& );
void square(const GAGenome& );
void DistanceCalc(const GAGenome& );
void setupMatrix(void);
float DistAIM( const GAGenome& );
float DistPAPM( const GAGenome& );
int   OXover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   INVover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   ROTover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   PATTover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   PCBover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   LASTover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
int   CROSSrecord(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
void tallyUpdate(const GAGenome& );
void resetLast(const GAGenome& );
void sumUp(const GAGenome&);
void resetTrailer(const GAGenome&);
void resetTally(void );

int gens = 0; //  A marker for tracking which generation is underway
int locgen = 1; //0=Read in all pcbdata; 1=Randomly generate such data
//Default is 1
int trad = 0; //1=not panelized (patternsV.txt);
//0=panelized (patterns.txt); Default is 1
int alike = 1;          //0=not identical pcb for random pcb component generation
```

```
                        //1=identical.  Default is 1.
int replc = 1; //Number of replications
int repeats = 1; //Number of GA repeated for same panel design
int popsize = 50; //  A default number of generations
float cx; //  Value which determines the crossover for given mating
float cmprsn = 0.; //Track comparator value
int duper = POPSIZE; //Counter for number of duplications in a generation
int fixed = 0; //  Indicator if all pcb are fixed (if so, = 1)
int probcheck = 0; //A flag to note that probabilities have beend checked
int marker[TAGS]; //  Temporary marker for specific crossover child
float score_list[MAXREPLC];//  Score list between GA replications
int templist[2*MAXREPLC][TOTALCOMPMAX];//A temporary holder for list values
int listsizes[2*MAXREPLC][TOTALCOMPMAX];//A holder for list sizes
int best_index = 0; //Track best score in score_list[]
int NN[TAGS]; //  Running tally of crossover operations
float mprob=1.; //  Prob of mutating a particular node within a string
float pcross = 1.0; //  An initial crossover value
float prob[TAGS] =  //  probability list; must be "integer" at first,
{ //  to allow proper check after determining
0.,//  problem type, listlength[] for 0 and 1
0.,
50.,
30.,
18.,
2.
};
float prob_save[TAGS] = //  Save the original probs for replications
{
0.,
0.,
0.,
0.,
0.,
0.
};
int dumpout = 0; //  A toggle for printing out crossover checks

int listnumber = 0; //    The sublist initializer tracker
//    used in initialization.
int listsize = 5; //   An arbitrary size; will be set later in
//   DesignInitializer and with listlength[]
int nlists = 5; //  Set at 5, and list 3 not used for AIM
int listlength[5]= //   An array of the lengths of each list
{
1,//   0:  this is always =1; only one pattern per solution
PCBMAX,//   1:  Number of PCB (rotation link)
TOTALCOMPMAX,//   2:  Component sequence
TOTALSLOTMAX,//   3:  Feeder allocation (could be reset if AIM)
TAGS //  4:  Tags which track the survivors' xovers
};

int pcbnum; //Total number of pcb
int racknum; //Total rack slots
int pattnum; //Total number of patterns
int comp_low, comp_hi; //Low/hi for components on any PCB
int ctyp_low, ctyp_hi; //Low/hi for no. of comptypes on panel
float bottom_clear = 0.; //Clearance from panel bottom
float slotspace = 0.; //Spacing between slots
float middlespace = 0.; //Middle of panel width
int heads = 12; //Number of heads for RTHM
float indextime = 0.15; //Index time (sec/component) for RTHM
float slottime = 0.15; //Feeder slot time (sec/slots moved) for RTHM
float tablevel = 1.0; //Table velocity (mm/sec) for RTHM
```

```
int innum = 0; //Arg for the input file; used for deterministic cases
int pcbb = 0; //Arg for number of pcb involved
int rect = 0; //Arg for WriteAcad: 1=print comp rectangles
//      0=print comp number (default)
int aim = 0; //Set aim case:  0 for no aim, 1 for aim
int papm = 1; //Set papm case: 0 for no papm, 1 for papm (default)
int rthm = 0; //Set rthm case:  0 for no rthm, 1 for rthm
//float rcx[TOTALSLOTMAX]; //Measure for centers of component population(s)
//float rcy[TOTALSLOTMAX]; //Measure for centers of component population(s)
float type[TOTALSLOTMAX]; //Component types recorded (as necessary)
float Rc[TOTALSLOTMAX]; //Magnitude of rx/y center(s)

int * aa0 = new int[PCBMAX];    // The rotation part 0; depends on the sublist[1]
int * aa1 = new int[PCBMAX]; // I did not redefine the array size within the
int * aa2 = new int[PCBMAX]; // program after the readFile#(), because these
int * aa3 = new int[PCBMAX]; // are arrays of pointers and don't take up too
// much space as a result.  Could do it later.
int * dups = new int[duper];    // An array which tracks the duplicates in a gen

float x[TOTALCOMPMAX]; //  Temporary x panel coords for placement sequence.
float y[TOTALCOMPMAX]; //  Temporary y panel coords for placement sequence.
float sx[TOTALSLOTMAX]; //  Temporary x panel coords for slot locations.
float sy[TOTALSLOTMAX]; //  Temporary y panel coords for slot locations.

float cdist[TOTALCOMPMAX][TOTALCOMPMAX]; //Static distance matrix for trad 1 innum 0
float sdist[TOTALCOMPMAX][TOTALCOMPMAX];//Static slot distance matrix for rthm;
//also used for trad 1 innum 0
int assign[TOTALSLOTMAX]; //Maps types to slot locations

int * sqrect = new int[PCBMAX]; //  (pointers to the) Shape array
float * PCBw = new float[PCBMAX]; //  (pointers to the) width array
float * PCBh = new float[PCBMAX]; //  (pointers to the) width array

char name0[40];
char name1[40];
char name2[40];
char name3[40];
char name4[40];
char name5[40];
char name6[40];
char name7[40];
char name8[40];
char acadname[40]; //Used to reset a new acad name
char acad_base[5]="acad"; //Base name for acad files
char inputf[9]="test"; //Base name for input files
char inputf2[9]="pcbV"; //Base name for input files
char input2f[10]="pattV"; //Base name for pattern input file
char input2f2[9]="patterns"; //Base name for pattern input file
char locs[5]="locs"; //Base name for location record files
char locations[10]="locations"; //Base name for location input files
char rates[6] = "rates"; //Base name for rates data
char check[6] = "check"; //Base name for output/check file
char pcb_char[3]; //character used for case filename

char innum_char[2]; //character used for case filename

FILE *infile;
FILE *outfile;                    //an output file

ofstream fout; //  Output function for results file
ofstream fcheck; //  Output function for check file
ofstream fprob; //  Output function for rates file
ofstream acadout;               //  Acad output file
ofstream fscore;                //  Output function for score file
```

```
ofstream flocs;                //  Output function for score file
ifstream fin; //  Input function for input file(not used)

//  Here come the input structures
struct components  //   Component structure
{
  int ct; //  Component type
  int pcbid; //  PCB identification type
  float xloc;  //  LOCAL (PCB-relative) x-location (was double)
  float yloc; //  LOCAL (PCB-relative) y-location(was double)
};

struct feeder   //   feeder (rack) structure
{
  int ct; //  Component type assigned
  float xglob; //  GLOBAL x-location(was double)
  float yglob; //  GLOBAL y-location(was double)
};

struct pattern   //   pattern structure
{
  float XX[PCBMAX]; //  PCB center GLOBAL x-location(was double)
  float YY[PCBMAX]; //  PCB center GLOBAL y-location(was double)
  int OO[PCBMAX]; //  PCB center GLOBAL orientation (degrees)
};

// A structure for storing results between replications
struct replication  //Replication structure
{
  float score; //Final replication score
  int panelcomps; //Total panel component population
  int paneltypes; //Total panel component types used
  int gens; //Generations require for replication
  int pcbcomp[PCBMAX]; //Array of components per pcb
};

//  NOTE that the following sets are POINTERS, not structure in themselves

components * comp = new components[TOTALCOMPMAX];//  component structure
feeder * slot = new feeder[TOTALSLOTMAX];//  feeder structure
pattern * patt = new pattern[PATTERNMAX];//  pattern structure
replication * results = new replication[repeats*replc];//Replication results

/**********************************************************************/
/*  Set alpha vector from rotation requirements                       */
/**********************************************************************/
void alphaSet( int kk, int alp ) {

        switch ( alp/90 )
        {

        // alp = 0 degrees
        case 0:
        aa0[kk] = 1;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // alp = 90 degrees
        case 1:
        aa0[kk] = 0;
        aa1[kk] = 0;
        aa2[kk] = 1;
```

```
        aa3[kk] = 0;
        break;

        // alp = 180 degrees
        case 2:
        aa0[kk] = 0;
        aa1[kk] = 1;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // alp = 270 degrees
        case 3:
        aa0[kk] = 0;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 1;
        break;

        // alp = 360 degrees
        case 4:
        aa0[kk] = 1;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // fall-out
        default:
        cout << "\nPCB " << kk << ", degree = " << alp << "\n";
        cout << "aa0["<<kk<<"]= "<<aa0[kk]<<  ", aa1["<<kk<<"]= "<<aa1[kk]
          << ", aa2["<<kk<<"]= "<<aa2[kk]<<  ", aa3["<<kk<<"]= "<<aa3[kk];
        cout << "\n\nSomething wrong with aa[] switching statement\n\n";
        exit(1);
        }
}

// This is the genetic algorithm that does the restricted mating.  It is
// similar to the steady state genetic algorithm, but we modify the selection
// part of the step method to do the restricted mating.

class CarefulMatingGA : public GASteadyStateGA {
public:
  GADefineIdentity("CarefulMatingGA", 288);
  CarefulMatingGA(const GAGenome& g) : GASteadyStateGA(g) {}
  virtual ~CarefulMatingGA() {}
  virtual void step();
  virtual void screen();
  virtual void cull();
  CarefulMatingGA & operator++() { cull(); return *this; }
};

// This step method is similar to that of the regular steady-state genetic
// algorithm, but we sift through aggregate pool and avoid duplicating identical
// members before cull.
// I've also separated the culling process into a separate function (::cull).
void
CarefulMatingGA::step() {
  int i, k, mut, c1, c2;
//  int psize;
  GAGenome *mom, *dad;          // tmp holders for selected genomes

//    This is the main event; tmpPop gathers the mating pairs,
// converts them to children, and adds them to pop.
```

```
  for(i=0; i<tmpPop->size()-1; i+=2){   // takes care of odd population
    mom = &(pop->select());
    int k = 0; //  Insure something is selected
    do{
      dad = &(pop->select());  k++;
      mom->compare(*dad); //  Set comparision tracker
    }while( cmprsn<1 && k<50);//  Make sure mom/dad not same
    stats.numsel += 2;          // keep track of number of selections


    c1 = c2 = 0;
    if(GAFlipCoin(pcross)){ //  Set cx here
      cx = GARandomFloat(0.0, pcross);
      stats.numcro += (*scross)(*mom, *dad, &tmpPop->individual(i),
                                &tmpPop->individual(i+1));
      c1 = c2 = 1;
    }
    else{
      tmpPop->individual( i ).copy(*mom);
      tmpPop->individual(i+1).copy(*dad);
      stats.nummut += (mut = tmpPop->individual( i ).mutate(mprob));
      if(mut > 0) c1 = 1;
      stats.nummut += (mut = tmpPop->individual(i+1).mutate(mprob));
      if(mut > 0) c2 = 1;

    }
   stats.numeval += c1 + c2;
  }
  if(tmpPop->size() % 2 != 0){  // do the remaining population member
    mom = &(pop->select());
    k = 0;
    do{
      dad = &(pop->select());  k++;
      mom->compare(*dad); //  Set comparision tracker
    }while( cmprsn < 1 && k<50 ); //  Make sure mom/dad not same
    stats.numsel += 2;          // keep track of number of selections

    c1 = 0;
    if(GAFlipCoin(pcross)){
      cx = GARandomFloat(0.0, pcross);
      stats.numcro += (*scross)(*mom, *dad,
                                &tmpPop->individual(i), (GAGenome*)0);
      c1 = 1;
    }
    else{
      if(GARandomBit())
        tmpPop->individual( i ).copy(*mom);
      else
        tmpPop->individual( i ).copy(*dad);
    stats.nummut += (mut = tmpPop->individual( i ).mutate(mprob));
    if(mut > 0) c1 = 1;

    }
  stats.numeval += c1;
  }

// We invoke the population's add member with a
// genome pointer rather than reference.  This way we don't force a clone of
// the genome - we just let the population take over.  This swells up the
// pop to greater than its usual popsize, but it will be reduced by cull.

  //  Output pop
  if(dumpout==3){
```

```
    fcheck << "\npop in step\n ";
    for(i=0; i<pop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << pop->individual(i);
      fcheck << pop->individual(i).score()<<"\n";
    }
  }
  //  Output tmpPop
  if(dumpout==3){
    fcheck << "\ntmpPop in step\n ";
    for(i=0; i<tmpPop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << tmpPop->individual(i);
      fcheck << tmpPop->individual(i).score()<<"\n";
    }
  }

  for(i=0; i<tmpPop->size(); i++){
    pop->add(&tmpPop->individual(i));
  }
  pop->evaluate();              // Prep for eval
  pop->scale();                 // Prep for scaling

  //  Output aggregate pop
  if(dumpout==3){
    fcheck << "\naggregatepopulation in step\n ";
    for(i=0; i<pop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << pop->individual(i);
      fcheck << pop->individual(i).score()<<"\n";
    }
  }
}

// Screen duplications in the aggregate population.
// Only one or two pop up per generation per a pop size of about 100, but
// that small number compounds the problem over thousands of generations.
void
CarefulMatingGA::screen() {
  int i,j, mut;


//Look through pop and see if there are duplicates.
  mut = 0;
  duper = 0;
  for(i=0; i<pop->size()-1; i++){
    for(j=i+1; j<pop->size(); j++){
      if(dumpout==7) fcheck<<"\nscreen compare of "<<i<<" and "<<j;
      pop->individual(i).compare( pop->individual(j) );
      if(cmprsn < 1){
        do{
  mut = pop->individual(j).mutate(mprob);
}while(mut<1);
dups[duper]=j; //Record which individual was a modified duplication
duper++; //Increment the number of modified duplications
mut = 0;
      }
    }
  }
}

// Cull out the worst individuals in the aggregate population
// Evaluate MUST be called before this is used to be effective
```

```
void
CarefulMatingGA::cull() {
  int i, mut;

  mut = 0;
  //  Output pop
  if(dumpout==3){
    fcheck << "\npop in cull\n ";
    for(i=0; i<pop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << pop->individual(i);
      fcheck << pop->individual(i).score()<<"\n";
    }
  }
  //  Output tmpPop
  if(dumpout==3){
    fcheck << "\ntmpPop in cull:\n ";
    for(i=0; i<tmpPop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << tmpPop->individual(i);
      fcheck << tmpPop->individual(i).score()<<"\n";
    }
  }
// The individuals in tmpPop are all owned by pop, but tmpPop does not know
// that.  so we use replace to take the individuals from the pop and stick
// them back into tmpPop

  pop->scale(gaTrue);
  pop->sort(gaTrue, GAPopulation::SCALED);

  for(i=0; i<tmpPop->size(); i++){
   tmpPop->replace(pop->remove(GAPopulation::WORST, GAPopulation::SCALED), i);
  }

  //  Output pop and tmpPop after replacement
  if(dumpout==3){
    fcheck << "\npopulation\n ";
      for(i=0; i<pop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << pop->individual(i);
      fcheck << pop->individual(i).score()<<"\n";
    }
    fcheck << "\ntmpPop \n";
    for(i=0; i<tmpPop->size(); i++){
      fcheck <<"Genome "<<i<<":\n";
      fcheck << tmpPop->individual(i);
      fcheck << tmpPop->individual(i).score()<<"\n";
    }
  }

  stats.numrep += tmpPop->size();

  stats.update(*pop);           // update the statistics by one generation

}

//  This is the class definition for the genome object.
//  Some of the methods of the base class are overridden.  The list
//  genomes in the composite genome are assigned the 'List' operators
//  by default.

class PanelDesignGenome : public GAGenome {
public:
  GADefineIdentity("PanelDesignGenome", 251);
```

```
  static void Initializer(GAGenome&);
  static int Mutator(GAGenome&, float);
  static float Comparator(const GAGenome&, const GAGenome&);
  static float Evaluator(GAGenome&);
  static void DesignInitializer(GAGenome&);
  static int ParseCrossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);

public:
  PanelDesignGenome(int nlists);
  PanelDesignGenome(const PanelDesignGenome & orig) {nn=0; list=0; copy(orig); }
  PanelDesignGenome operator=(const GAGenome & arg) { copy(arg); return *this; }
  virtual ~PanelDesignGenome();
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const;
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;
  GAListGenome<int> & path(const int i){return *list[i];}
  int npaths() const { return nn; }

protected:
  int nn;
  GAListGenome<int> **list;
};

PanelDesignGenome::PanelDesignGenome(int nlists) :
GAGenome(Initializer, Mutator, Comparator){
  evaluator(Evaluator);
  crossover(ParseCrossover);
  nn = nlists;  //  This is a bit of a cheat; nlists is GLOBAL
//  But, it is valuable with the option of
//  changing the nlists in main, below.
  list = (nn ? new GAListGenome<int> * [nn] : (GAListGenome<int> **)0);

  for(int i=0; i<nn; i++){
    list[i] = new GAListGenome<int>;
    list[i]->initializer(DesignInitializer);
    list[i]->mutator(GAListGenome<int>::SwapMutator);
  }

}

void
PanelDesignGenome::copy(const GAGenome& g) {
  if(&g != this && sameClass(g)){
    GAGenome::copy(g); // copy the base class part
    PanelDesignGenome & genome = (PanelDesignGenome &)g;
    if(nn == genome.nn){
      for(int i=0; i<nn; i++) //   Note the global "nn" here
list[i]->copy(*genome.list[i]);
    }
    else{
      int i;
      for(i=0; i<nn; i++) //   Note the global "nn" here
delete list[i];
      delete [] list;
      nn = genome.nn;  //   Note the global "nn" here
      list = new GAListGenome<int> * [nn];
      for(i=0; i<nn; i++)
list[i] = (GAListGenome<int> *)genome.list[i]->clone();
    }
  }
}
```

```
PanelDesignGenome::~PanelDesignGenome(){
  for(int i=0; i<nn; i++)  //   Note the global "nn" here
    delete list[i];
  delete [] list;
}

GAGenome*
PanelDesignGenome::clone(GAGenome::CloneMethod) const {
  return new PanelDesignGenome(*this);
}

int
PanelDesignGenome::equal(const GAGenome& g) const {
  PanelDesignGenome& genome = (PanelDesignGenome&)g;
  int flag=0;
  for(int i=0; i<nn && flag==0; i++)  //   Note the global "nn" here
    flag = list[i]->equal(*genome.list[i]);
  return flag;
}

int
PanelDesignGenome::read(istream & is) {
  for(int i=0; i<nn; i++) //   Note the global "nn" here
    is >> *(list[i]);
  return is.fail() ? 1 : 0;
}

int
PanelDesignGenome::write(ostream & os) const {
  for(int i=0; i<nn; i++) //   Note the global "nn" here
    os << "list " << i << ":\t" << *(list[i]) << "\n";
  return os.fail() ? 1 : 0;
}

//  Operators are set up below
void
PanelDesignGenome::Initializer(GAGenome& g) {
  PanelDesignGenome & genome = (PanelDesignGenome &)g;

  for(int i=0; i<genome.npaths(); i++){

    listnumber = i; //    Tracking which sublist (initialization)
    listsize = listlength[i]; //    Differently sized links

    genome.path(i).initialize();

    //     cout<<"\nInside the Initializer\n"; //
    //     cout<<"genome.path("<<i<<")= "<<genome.path(i)<<"\n"; //
  }

}

//   Below, we can just chop out the mutations for the first two links,
//   representing the patterns and the pcbtypes.
//   Do this by starting 'for' loop from i=2
//  Also, end the loop "early" because of the trailing tracker sublist.
int
PanelDesignGenome::Mutator(GAGenome& g, float pmut) {
  PanelDesignGenome & genome = (PanelDesignGenome &)g;
  GAListGenome<int> &sublist=(GAListGenome<int> &)genome.path(nlists-1);
  int nMut = 0;  int i = 0;
  int endit = genome.npaths()-1;
  if(aim) endit = genome.npaths()-2;
  for(i=2; i<endit; i++)
```

```
    nMut += genome.path(i).mutate(pmut);

  //Change trailer list value to indicate mutation, resetting former value
  for(i=0; i<TAGS; i++) *sublist.warp(i) = 0;
  *sublist.warp(TAGS-1) = 1; //This is the mutation tag

  return nMut;
}

//  Use this function to screen out identical population members in ::step()
float
PanelDesignGenome::Comparator(const GAGenome& a, const GAGenome& b) {
  PanelDesignGenome& sis = (PanelDesignGenome &)a;
  PanelDesignGenome& bro = (PanelDesignGenome &)b;
  float diff = 0;
  for(int i=0; i<sis.npaths()-1; i++){
    diff += sis.path(i).compare(bro.path(i));
  }
    cmprsn = diff;
  if(dumpout==7){
           fcheck << "\ncmprsn = " << cmprsn ;
   fcheck << "\nfirst:\n"<< sis;
   fcheck << "\nsecond:\n"<< bro;
   }
  return diff;
}

//Set up the distance matrix if required
//Note that there is only one pattern, so just reference patt[0] directly
//Also, x[] and y[] are never needed again, so they can be used later in
//the program (and are:  They are used to plot solutions by AcadWrite)
void
setupMatrix(void)
{
  int i,j,k;

  //   Get the x,y for components in sequence order and in

  //Call up alphas.  Necessary in case the pattern has .OO != 0.
  for (k=0; k<listlength[1]; k++) alphaSet( k, patt[0].OO[k] );

  //   Get the x,y for components in CARDINAL order and in
  //   panel frame of reference.
  //This is different than for the all-patterns, all-rotations case.
    for (i=0; i<listlength[2]; i++) {
      x[i] = patt[0].XX[ comp[i].pcbid ]
     + comp[i].xloc *
aa0[ comp[i].pcbid ]
     - comp[i].xloc *
aa1[ comp[i].pcbid ]
     - comp[i].yloc *
aa2[ comp[i].pcbid ]
     + comp[i].yloc *
aa3[ comp[i].pcbid ] ;

      y[i] = patt[ 0 ].YY[ comp[i].pcbid ]
     + comp[i].yloc *
aa0[ comp[i].pcbid ]
     - comp[i].yloc *
aa1[ comp[i].pcbid ]
     + comp[i].xloc *
aa2[ comp[i].pcbid ]
     - comp[i].xloc *
aa3[ comp[i].pcbid ] ;
```

```
    }

  //PAPM and RTHM have slot distances to contend with.  PAPM deals with
  //them in the cdist[] matrix, but RTHM has its own special distance
  //matrix, though it is really simple (all are equally spaced, or should be)
  if( rthm){
      for(i=0; i<listlength[3]; i++)
        for(j=0; j<listlength[3]; j++)
  sdist[i][j] = sqrt( pow(slot[i].xglob - slot[j].xglob, 2 )
                   + pow(slot[i].yglob - slot[j].yglob, 2 ) );
  }
  //If AIM or RTHM, the x,y are needed for the dist matrix
  if(aim || rthm){
    for(i=0; i< listlength[2]; i++)
      for(j=0; j< listlength[2]; j++)
cdist[i][j] = sqrt( pow( x[i] - x[j], 2) + pow( y[i] - y[j], 2) );
  }
  //For papm, dist is distance from component i to slotLOCATION j.
  else if(papm){
    for(i=0; i< listlength[2]; i++)
      for(j=0; j< listlength[3]; j++)
cdist[i][j] = sqrt( pow( x[i] - slot[j].xglob, 2)
                + pow( y[i] - slot[j].yglob, 2) );
  }
}

//  The distance for a solution calculations.  x,y,sx,xy are kept as global.
void
DistanceCalc(const GAGenome& p){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;

  int i,j,k;
  int degree = 0; //  A placekeeper for PCB rotations (convenience)
  //    CALL alphaSet to get right aa#[] for this particular solution.
  for (k=0; k<listlength[1]; k++){
    degree = *gen.path(1).warp(k) + patt[*gen.path(0).warp(0)].OO[k];
    alphaSet( k, degree ); //
  }
  //   Slots are ordered in terms of component types assigned.
  //   Need to get (sx,sy) coords in order of component types.
  //   (Not really necessary, but looks neater in Dist??? subroutines)

  //    Reading left to right:
  //  Value for component type j in a sublist position is assigned
  //  to slot position in order.
  //  SO, think like this:
  //   sx[component type at sublist position] = slot[sublist postion].x/y
  if(!aim){
    for (j=0; j<listlength[3]; j++){
      sx[*gen.path(3).warp(j)] = slot[j].xglob;
      sy[*gen.path(3).warp(j)] = slot[j].yglob;
    }
  }

  //   Get the x,y for components in sequence order and in
  //   panel frame of reference.
    for (i=0; i<listlength[2]; i++) {
      x[i] = patt[*gen.path(0).warp(0)].XX[ comp[*gen.path(2).warp(i)].pcbid ]
     + comp[*gen.path(2).warp(i)].xloc *
aa0[ comp[*gen.path(2).warp(i)].pcbid ]
     - comp[*gen.path(2).warp(i)].xloc *
aa1[ comp[*gen.path(2).warp(i)].pcbid ]
     - comp[*gen.path(2).warp(i)].yloc *
aa2[ comp[*gen.path(2).warp(i)].pcbid ]
```

```
     + comp[*gen.path(2).warp(i)].yloc *
aa3[ comp[*gen.path(2).warp(i)].pcbid ] ;

      y[i] = patt[ *gen.path(0).warp(0) ].YY[ comp[*gen.path(2).warp(i)].pcbid ]
     + comp[*gen.path(2).warp(i)].yloc *
aa0[ comp[*gen.path(2).warp(i)].pcbid ]
     - comp[*gen.path(2).warp(i)].yloc *
aa1[ comp[*gen.path(2).warp(i)].pcbid ]
     + comp[*gen.path(2).warp(i)].xloc *
aa2[ comp[*gen.path(2).warp(i)].pcbid ]
     - comp[*gen.path(2).warp(i)].xloc *
aa3[ comp[*gen.path(2).warp(i)].pcbid ] ;

    }

}

// This subroutine calculates distances for the AIM problem
// with the static distance matrix, so MUCH faster
float
DistAIMTrad( const GAGenome& p ){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;
  float dist = 0;
  for (int i=0; i<listlength[2]-1; i++)
    dist += cdist[*gen.path(2).warp(i)][*gen.path(2).warp(i+1)];
  //Tack on the distance to close TSP
  dist += cdist[*gen.path(2).warp(0)][*gen.path(2).warp(listlength[2]-1)];
  return dist;
}

// This subroutine calculates distances for the PAPM problem
// with the static distance matrix, so MUCH faster
float
DistPAPMTrad( const GAGenome& p ){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;
  float dist = 0.;
  int i,j;

  //This must be done for every call.
  //Slots are ordered in terms of component types assigned.
  //Need to get slot coords in order of component types.
    for (j=0; j<listlength[3]; j++){
      assign[*gen.path(3).warp(j)] = j;
    }

  //  It is simplest if separated into "required" and "non-required" moves
  //First the slot-to-component distance... "required"
  for (i=0; i<listlength[2]; i++)
    dist += cdist[*gen.path(2).warp(i)][assign[comp[*gen.path(2).warp(i)].ct]];

   //Then the component-to-slot distance... "non-required"
  for (i=0; i<listlength[2]-1; i++) {
    dist += cdist[*gen.path(2).warp(i)][assign[comp[*gen.path(2).warp(i+1)].ct]];
  }

  //Add in the first move from last slot of last panel to first component
    dist += cdist[*gen.path(2).warp(listlength[2]-1)][assign[comp[*gen.path(2).warp(0)].ct]];

  return dist;

}

// This subroutine calculates distances for the RTHM problem
// with the static distance matrix, so MUCH faster
```

```
//NOTE:  The returned value is in seconds, not mm (as in AIM and PAPM)
float
DistRTHMTrad( const GAGenome& p ){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;
  float dummytime = 0.0; //Dummy time keeper in loop
  float totaltime = 0.0; //Total time returned
  float tabletime = 0.0; //Table time between placements
  float feedertime = 0.0; //Feeder movement between placements
  int i,j;
  int k = 0;

  int leftover = heads/2; //Number ofcomponents left to place
//for next panel in batch

  //This must be done for every call.
  //Slots are ordered in terms of component types assigned.
  //Need to get slot coords in order of component types.
    for (j=0; j<listlength[3]; j++){
      assign[*gen.path(3).warp(j)] = j;
    }
  //Get times up to total panel component count minus the leftover
  for (i=1; i<listlength[2]-leftover; i++) {
    //Time to move to next component location on panel
    tabletime = cdist[*gen.path(2).warp(i)][*gen.path(2).warp(i-1)]/ tablevel;

    //Time to move from one feeder slot to next;
    //Distances have been used, so divide by distances between slots
    //and multiply by slottime (per slot moved)
    feedertime = (slottime/slotspace ) *
sdist[assign[comp[*gen.path(2).warp(i+leftover)].ct]]
     [assign[comp[*gen.path(2).warp(i+leftover-1)].ct]];

    //Noting that there is always the indextime between placements,
    //choose the maximum time required
    dummytime =  feedertime > tabletime ? feedertime : tabletime;
    dummytime =  dummytime  > indextime ?  dummytime : indextime;
    totaltime = totaltime + dummytime;
  }

  //Now, the i counter is at listlength[2]-leftover.  We have this component
  //through to listlength[2]-1 to place.  Assume that the next 0 through
  //leftover components are to be picked up during the last leftover placments

  //First, get from the ith component to i+1, noting that you are now picking up
  //from 0 to 0+1.

    //Time to move to next component location on panel
    tabletime = cdist[*gen.path(2).warp(i)][*gen.path(2).warp(i-1)]/ tablevel;

    //Time to move from one feeder slot to next;
    feedertime = (slottime/slotspace ) *
sdist[assign[comp[*gen.path(2).warp(0)].ct]]
     [assign[comp[*gen.path(2).warp(listlength[2]-1)].ct]];

    //choose the maximum time required
    dummytime =  feedertime > tabletime ? feedertime : tabletime;
    dummytime =  dummytime  > indextime ?  dummytime : indextime;
    totaltime = totaltime + dummytime;

  i++; //Advance to the next component

  k=1; //Need to start at 1 here
  //Then, continue around the head...
  for(j=i; j<listlength[2]; j++){
```

```
    //Time to move to next component location on panel
    tabletime = cdist[*gen.path(2).warp(j)][*gen.path(2).warp(j-1)]/ tablevel;

    //Time to move from one feeder slot to next;
    feedertime = (slottime/slotspace ) *
sdist[assign[comp[*gen.path(2).warp(k)].ct]]
     [assign[comp[*gen.path(2).warp(k-1)].ct]];

    //choose the maximum time required
    dummytime =  feedertime > tabletime ? feedertime : tabletime;
    dummytime =  dummytime  > indextime ?  dummytime : indextime;
    totaltime = totaltime + dummytime;

    k++; //increment k
  }

    //And close the loop back to the start.
    k++; //increment k
    tabletime = cdist[*gen.path(2).warp(0)][*gen.path(2).warp(listlength[2]-1)]
                / tablevel;
    feedertime = (slottime/slotspace ) *
sdist[assign[comp[*gen.path(2).warp(k)].ct]]
     [assign[comp[*gen.path(2).warp(k-1)].ct]];
    //choose the maximum time required
    dummytime =  feedertime > tabletime ? feedertime : tabletime;
    dummytime =  dummytime  > indextime ?  dummytime : indextime;
    totaltime = totaltime + dummytime;

  return totaltime;

}

// This subroutine calculates distances for the AIM problem
// for the full panelization implementation.
// Note that x,y are already in sequence order
float
DistAIM( void ){
  float dist = 0;
  for (int i=1; i<listlength[2]; i++) {
    dist = dist + sqrt( pow( ( x[i]-x[i-1] ),2 ) + pow( ( y[i]-y[i-1] ),2 ) );
  }
  // Added a line to "close the loop" on the TSP
  dist = dist + sqrt( pow( ( x[0]-x[listlength[2]-1] ),2 )
                + pow( ( y[0]-y[listlength[2]-1] ),2 ) );
  return dist;
}

// This subroutine calculates distances for the PAPM problem
float
DistPAPM( const GAGenome& p ){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;
  float dist = 0.;
  int i;

  //  It is simplest if separated into "required" and "non-required" moves
  //First the slot-to-component distance... "required"
  for (i=0; i<listlength[2]; i++) {
    dist = dist +
      sqrt( pow( ( sx[comp[*gen.path(2).warp(i)].ct] - x[i] ),2 )
+ pow( ( sy[comp[*gen.path(2).warp(i)].ct] - y[i] ),2 ) );
  }

   //Then the component-to-slot distance... "non-required"
   //Note that the last component-to-nextslot distance is assumed unnecessary
```

```
   //because placement process is finished at placement of last component
  for (i=0; i<listlength[2]-1; i++) {
    dist = dist +
      sqrt( pow( ( x[i] - sx[comp[*gen.path(2).warp(i+1)].ct ] ),2 )
+ pow( ( y[i] - sy[comp[*gen.path(2).warp(i+1)].ct ] ),2 ) );
  }

  //  fcheck << "\nDistPAPM non-required dist = " << dist << "\n";

  return dist;

}

// This subroutine calculates distances for the RTHM problem
//NOTE:  The returned value is in seconds, not mm (as in AIM and PAPM)
float
DistRTHM( const GAGenome& p ){
  PanelDesignGenome &gen=(PanelDesignGenome &)p;
  float dummytime = 0.0; //Dummy time keeper in loop
  float totaltime = 0.0; //Total time returned
  float tabletime = 0.0; //Table time between placements
  float feedertime = 0.0; //Feeder movement between placements
  int i,j;
  int k = 0;

  int leftover = heads/2; //Number ofcomponents left to place
//for next panel in batch

  //Get times up to total panel component count minus the leftover
  for (i=1; i<listlength[2]-leftover; i++) {
    //Time to move to next component location on panel
    tabletime =
      sqrt( pow( ( x[i]-x[i-1] ),2 )
    + pow( ( y[i]-y[i-1] ),2 ) ) / tablevel;

    //Time to move from one feeder slot to next;
    //Distances have been used, so divide by distances between slots
    //and multiply by slottime (per slot moved)
    feedertime = slottime * sqrt(
        pow( fabs(
          sx[comp[*gen.path(2).warp(i+leftover)].ct ]
        - sx[comp[*gen.path(2).warp(i+leftover-1)].ct ] ), 2 )
      + pow( fabs(
          sy[comp[*gen.path(2).warp(i+leftover)].ct ]
        - sy[comp[*gen.path(2).warp(i+leftover-1)].ct ] ), 2 )
        ) / slotspace;
    //Noting that there is always the indextime between placements,
    //choose the maximum time required
    dummytime =  feedertime > tabletime ? feedertime : tabletime;
    dummytime =  dummytime  > indextime ?  dummytime : indextime;
    totaltime = totaltime + dummytime;

  }

  //Now, the i counter is at listlength[2]-leftover.  We have this component
  //through to listlength[2]-1 to place.  Assume that the next 0 through
  //leftover components are to be picked up during the last leftover placments

  //First, get from the ith component to i+1, noting that you are now picking up
  //from 0 to 0+1.

    //Time to move to next component location on panel
    tabletime =
      sqrt( pow( ( x[i]-x[i-1] ),2 )
```

```
  + pow( ( y[i]-y[i-1] ),2 ) ) / tablevel;

  //Time to move from one feeder slot to next;
  feedertime = slottime * sqrt(
      pow( fabs(
        sx[comp[*gen.path(2).warp(0)].ct ]
      - sx[comp[*gen.path(2).warp(listlength[2]-1)].ct ] ), 2 )
    + pow( fabs(
        sy[comp[*gen.path(2).warp(0)].ct ]
      - sy[comp[*gen.path(2).warp(listlength[2]-1)].ct ] ), 2 )
      ) / slotspace;

  //choose the maximum time required
  dummytime =  feedertime > tabletime ? feedertime : tabletime;
  dummytime =  dummytime  > indextime ?  dummytime : indextime;
  totaltime = totaltime + dummytime;

i++; //Advance to the next component

k=1; //Need to start at 1
//Then, finish up after the transition
for(j=i; j<listlength[2]; j++){
  //Time to move to next component location on panel
  tabletime =
    sqrt( pow( ( x[j]-x[j-1] ),2 )
  + pow( ( y[j]-y[j-1] ),2 ) ) / tablevel;

  //Time to move from one feeder slot to next;
  feedertime = slottime * sqrt(
      pow( fabs(
        sx[comp[*gen.path(2).warp(k)].ct ]
      - sx[comp[*gen.path(2).warp(k-1)].ct ] ), 2 )
    + pow( fabs(
        sy[comp[*gen.path(2).warp(k)].ct ]
      - sy[comp[*gen.path(2).warp(k-1)].ct ] ), 2 )
      ) / slotspace;

  //choose the maximum time required
  dummytime =  feedertime > tabletime ? feedertime : tabletime;
  dummytime =  dummytime  > indextime ?  dummytime : indextime;
  totaltime = totaltime + dummytime;

  k++; //increment k
}

  //And close the loop back to the start.
  k++; //increment k
  tabletime =
    sqrt( pow( ( x[0]-x[listlength[2]-1] ),2 )
  + pow( ( y[0]-y[listlength[2]-1] ),2 ) ) / tablevel;
  //Time to move from one feeder slot to next;
  feedertime = slottime * sqrt(
      pow( fabs(
        sx[comp[*gen.path(2).warp(k)].ct ]
      - sx[comp[*gen.path(2).warp(k-1)].ct ] ), 2 )
    + pow( fabs(
        sy[comp[*gen.path(2).warp(k)].ct ]
      - sy[comp[*gen.path(2).warp(k-1)].ct ] ), 2 )
      ) / slotspace;
  //choose the maximum time required
  dummytime =  feedertime > tabletime ? feedertime : tabletime;
  dummytime =  dummytime  > indextime ?  dummytime : indextime;
  totaltime = totaltime + dummytime;
```

```
  return totaltime;

}

//  The objective function evaluates the genomes.
float
PanelDesignGenome::Evaluator(GAGenome & c) {
  PanelDesignGenome & genome = (PanelDesignGenome &)c;
  float score=0;

  if(trad) {
    if(aim) score = DistAIMTrad(genome);
    else if(papm) score = DistPAPMTrad(genome );
    else if(rthm) score = DistRTHMTrad(genome );
    else {
      cout << "\nNo machine defined:\naim = " << aim
           << "\tpapm = " << papm << "\trthm = " << rthm << "\n";
      exit(1);
    }
  }
  else {
    DistanceCalc(genome);

    if(aim) score = DistAIM( );
    else if(papm) score = DistPAPM(genome );
    else if(rthm) score = DistRTHM(genome );
    else {
      cout << "\nNo machine defined:\naim = " << aim
           << "\tpapm = " << papm << "\trthm = " << rthm << "\n";
      exit(1);
    }
  }

  return score;
}

//  This is needed in the OXover
//  It's also used in Ordered Crossover, so I use the name
//  GAListIsHole2 to avoid conflicts.

int GAListIsHole2(const GAListGenome<int> &child, const GAListGenome<int> &parent,
            int index, int a, int b){
  GAListIter<int> citer(child), piter(parent);
  citer.warp(index);
  piter.warp(a);
  for(int i=a; i<b; i++){
    if(*citer.current() == *piter.current()) return 1;
    piter.next();
  }
  return 0;
}


//  The OXover crossover function, used by only the sublists 2 and 3
//  OXover this is the OrderCrossover subroutine
//  This is the OX crossover in Goldberg (1987)
//  Keeps the "relative" positioning of the cities in order after crossover
//  by shifting the "holes" together produced from the crossover.

int
OXover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;
```

```
  int a,b,k;

//    Print out each genome every time OXover is accessed
    if( dumpout == 9 ) {
      fcheck << "\n OXover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
    }

  if(mom.size() != dad.size()){
    GAErr(GA_LOC, mom.className(), "cross", gaErrBadParentLength);
    return 0;
  }
  k=0;
  a = GARandomInt(0, mom.size());   //  Select a site (between nodes)
  do{
    b = GARandomInt(0, dad.size());   //  for each parent
    k++;
  }while(b==a && k<50);
  if(b<a) SWAP2(a,b);                   //  You want the dad site to be bigger
  int i,j, index, nc=0;                 //  (or equal) so just swap them

  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom
    GAListIter<int> siter(sis);         //  Define iteration pointers (?) for
    GAListIter<int> diter(dad);         //  both mom and dad

// Move all the 'holes' into the crossover section and maintain the ordering of
// the non-hole elements.
    for(i=0, index=b; i<sis.size(); i++, index++){
      if(index >= sis.size()) index=0;  //  This wraps dad's checking site value
      if(GAListIsHole2(sis,dad,index,a,b)) break;//  when you FIND sis()=dad()
    }                                           //  at some index, break and keep index

    for(; i<sis.size()-b+a; i++, index++){
      if(index >= sis.size()) index=0;
      j=index;
      do{
        j++;
        if(j >= sis.size()) j=0;
      } while(GAListIsHole2(sis,dad,j,a,b));
      sis.swap(index,j);
    }

// Now put the 'holes' in the proper order within the crossover section.
    for(i=a, sis.warp(a), diter.warp(a);
        i<b; i++, sis.next(), diter.next()){
      if(*sis.current() != *diter.current()){
        siter.warp(i);
        for(j=i+1; j<b; j++)
          if(*siter.next() == *diter.current()){
            sis.swap(i,j);
            sis.warp(siter);    // move iterator back to previous location
            break;
          }
      }
    }
    sis.head();         // set iterator to head of list
    nc += 1;

//    Print out each genome every time OXover is accessed
    if( dumpout == 9 ) {
      fcheck << "\nsis:  " << sis ;   //
```

```
    }

  }

  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);
    GAListIter<int> biter(bro);
    GAListIter<int> miter(mom);

// Move all the 'holes' into the crossover section and maintain the ordering of
// the non-hole elements.
    for(i=0, index=b; i<bro.size(); i++, index++){
      if(index >= bro.size()) index=0;
      if(GAListIsHole2(bro,mom,index,a,b)) break;
    }

    for(; i<bro.size()-b+a; i++, index++){
      if(index >= bro.size()) index=0;
      j=index;
      do{

        j++;
        if(j >= bro.size()) j=0;
      } while(GAListIsHole2(bro,mom,j,a,b));
      bro.swap(index,j);
    }

// Now put the 'holes' in the proper order within the crossover section.
    for(i=a, bro.warp(a), miter.warp(a);
        i<b; i++, bro.next(), miter.next()){
      if(*bro.current() != *miter.current()){
        biter.warp(i);
        for(j=i+1; j<b; j++)
          if(*biter.next() == *miter.current()){
            bro.swap(i,j);
            bro.warp(biter);    // move iterator back to previous location
            break;
          }
      }
    }
    bro.head();         // set iterator to head of list
    nc += 1;
//    Print out each genome every time OXover is accessed
    if( dumpout == 9 ) {
      fcheck << "\nbro:  " << bro ;   //
    }
  }

  return nc;
}

//   End of OXover program

//  Start of INVover program

//  This is the inversion of each parent selected.
//  Goldberg (1987)  points out that it only needs one parent,
//  but since two parents are selected as part of the overall GA
//  scheme, I'll invert both using different inversion point pairs.

int
INVover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
```

```
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

//    Print out each genome every time INVover is accessed
    if( dumpout==8 ) {
      fcheck << "\n INVover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
    }

  if(mom.size() != dad.size()){
    GAErr(GA_LOC, mom.className(), "cross", gaErrBadParentLength);
    return 0;
  }

  int a,b,i,n,flipper;
  int nc=0;

//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom

  n = mom.size();

  //  If only size 3 or 2, then invert from a to zero
  if(n < 4) {
     sis.head();         // set iterator to head of list
     if(n > 0){
       a = GARandomInt(0, n-1); //  Select a site (between nodes)
       b = 0;
       SWAP2(a,b);
     } //  If size == 0, then just return copy
     nc += 1;            //  Return number of childern (eventually)
     }
  else {

  a = GARandomInt(0, n-1);              //  Select a site (between nodes)
  b = GARandomInt(0, n-1);              //  Select a second site (between nodes)

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<" b:  "<< b <<"\n "; //

//  Want a gap between sites of at least 2, so...
  while (a == b || b-a == 1 || a-b == 1) b = GARandomInt(0, n-1);
  if(b<a) SWAP2(a,b);                   //  You want the 2nd site to be bigger
                                        //  so just swap them

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<" b:  "<< b <<"\n "; //


  if( ((b-a)%2) != 0 ) flipper = (b-a+1)/2 ; //  If odd, swap with a hole in middle
    else flipper = (b-a)/2 ;                  //  If diff is even, o.k.

  if( dumpout == 8 )fcheck<<"\n flipper:  "<< flipper <<"\n";

  for(i=0; i<flipper; i++, a++, b--)
    sis.swap(a,b);                              //  swap pointers between sites

//    Print out each genome every time INVover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n sis after inversion: \n sis:  " << sis <<"\n";//
      fcheck << "\n c1 after inversion: \n c1:  " << *c1 <<"\n";//
    }
```

```
    sis.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }
  }

//  "Bro" is based on dad...
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);         //  Make a copy of the dad


  n = dad.size();

  //  If only size 3, then just return duplicate of mother as child
  if(dad.size() < 4) {
     dad.head();         // set iterator to head of list
     if(n > 0){
       a = GARandomInt(0, n-1); //  Select a site (between nodes)
       b = 0;
       SWAP2(a,b);
     } //  If size == 0, then just return copy
     nc += 1;            //  Return number of childern (eventually)
     }
  else {

  a = GARandomInt(0, n-1);              //  Select a site (between nodes)
  b = GARandomInt(0, n-1);              //  Select a site (between nodes)

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<" b:  "<< b <<"\n "; //

//  Want a gap between sites of at least 2, so...
  while (a == b || b-a == 1 || a-b == 1) b = GARandomInt(0, n-1);
  if(b<a) SWAP2(a,b);                   //  You want the 2nd site to be bigger

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<" b:  "<< b <<"\n "; //

  if( (b-a)%2 != 0 ) flipper = (b-a+1)/2 ; //  If odd, swap with a hole in middle
    else flipper = (b-a)/2 ;                    //  If diff is even, o.k.

  if( dumpout == 8 )fcheck<<"\n flipper:  "<<flipper<<"\n";    //

  for(i=0; i<flipper; i++, a++, b--)
    bro.swap(a,b);                              //  swap pointers between sites

  if( dumpout == 8 )fcheck<<"\n flipper:  "<< flipper <<"\n";

//    Print out each genome every time INVover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n bro after inversion: \n bro:  " << bro <<"\n";//
      fcheck << "\n c2 after inversion: \n c2:  " << *c2 <<"\n";//
    }

    bro.head();         // set iterator to head of list
    nc += 1;
  }
  }

  if( dumpout == 8 ) {
    fcheck << "\n c1 :  " << *c1 ;     //  Write out first child results
    fcheck << "\n c2 :  " << *c2 ;     //  Write out second child results
  }
  return nc;
}
```

```
//  End of INVover (INVERSION)

//  Beginning of ROTover
//  This operator shifts nodes "a" through size-1 (the end node)
//  to the left and puts the front nodes after size-1

int
ROTover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

//    Print out each genome every time ROTover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n ROTover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
    }

  if(mom.size() != dad.size()){
    GAErr(GA_LOC, mom.className(), "cross", gaErrBadParentLength);
    return 0;
  }

  int a,b,n,i,j;
  int nc=0;

//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom

  n = mom.size();

  //  If only size 3, then just return duplicate of mother as child
  if(mom.size() < 4) {
     sis.head();         // set iterator to head of list
     if(n > 0){
       a = GARandomInt(0, n-1); //  Select a site (between nodes)
       b = 0;
       SWAP2(a,b);
     } //  If size == 0, then just return copy
     nc += 1;            //  Return number of childern (eventually)
     }
  else {

  a = GARandomInt(1, n-1);              //  Select a site (between nodes)
                                        //  also, we don't want a 0 node

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<"\n ";       //

  for(j=0; j<a; j++){                           //  Repeat til a node hits head
    for(i=0; i<n; i++){
    sis.swap(i,i+1);                            //  Swap pointers between sites
    }                                           //  and iterate through to end
  }                                             //

//    Print out each genome every time ROTover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n sis after inversion: \n sis:  " << sis <<"\n";//
    }

    sis.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
```

```
  }
  }

//  "bro" is based on dad...
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);                //  Make a copy of the dad

  n = dad.size();

  //  If only size 3, then just return duplicate of mother as child
  if(dad.size() < 4) {
     dad.head();         // set iterator to head of list
     if(n > 0){
       a = GARandomInt(0, n-1); //  Select a site (between nodes)
       b = 0;
       SWAP2(a,b);
     } //  If size == 0, then just return copy
     nc += 1;            //  Return number of childern (eventually)
     }
  else {

  a = GARandomInt(1, n-1);              //  Select a site (between nodes)
                                        //  also, we don't want a 0 node

  if( dumpout == 8 )fcheck<<"\n a:  "<<a<<"\n ";       //

  for(j=0; j<a; j++){                           //  Repeat til a node hits head
    for(i=0; i<n; i++){
    bro.swap(i,i+1);                            //  Swap pointers between sites
    }                                           //  and iterate through to end
  }                                             //

//    Print out each genome every time ROTover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n bro after inversion: \n bro:  " << bro <<"\n";//
    }

    bro.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }
  }
  return nc;
}

//  End of ROTover

//  Beginning of PCBover
//  PCBover merely calls the rectangle() or square() subs as appropriate
//  to change a single gene in the sublist.
//  Only one parent is manipulated at a time, but both have potential to
//  produce a child with swapped genes.

int
PCBover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

//    Print out each genome every time PCBover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n PCBover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
```

```
    }

  int a,n;
  int nc=0;

//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom

    n = mom.size();
    a = GARandomInt(0, n-1);              //  Select a site

    sis.warp(a); //  Warp to node for change
       if(sqrect[a] == 1) rectangle(sis); //  Change for rect
       if(sqrect[a] == 2) square(sis); //  Change for square
//  Note that a 0 is no change

    sis.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }

//  "bro" is based on dad...
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);                 //  Make a copy of the dad

  n = dad.size();
  a = GARandomInt(0, n-1);              //  Select a site

    bro.warp(a); //  Warp to node for change
       if(sqrect[a] == 1) rectangle(bro); //  Change for rectangle
       if(sqrect[a] == 2) square(bro); //  Change for square

    bro.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }
  return nc;

}

//  End of PCBover

//  Beginning of PATTover
//  PATTover merely swaps two positions or just sends back two identical
//  children, depending upon ppatt
//  Only one parent is manipulated at a time, but both have potential to
//  produce a child with swapped genes.
int
PATTover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

//    Print out each genome every time PATTover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n PATTover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
    }

  int a,b;
  int nc=0;
  int tries;
```

```
//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom

  tries = 0;
  b = *sis.head(); //Old value
  do{ a = GARandomInt(0,pattnum-1); // A new random value
      tries++;
    }while(a==b && tries < 50); //Don't repeat yourself, if possible

  *sis.head() = a;

  nc += 1;            //  Return number of childern (eventually)
  }

//  "bro" is based on dad...
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);                 //  Make a copy of the dad

  tries = 0;
  b = *bro.head(); //Old value
  do{ a = GARandomInt(0,pattnum-1); // A new random value
      tries++;
    }while(a==b && tries < 50); //Don't repeat yourself, if possible

  *bro.head() = a;

  nc += 1;            //  Return number of childern (eventually)
  }
  return nc;

}

//  End of PATTover

//  Beginning of CROSSover
//  CROSSover returns copies of the parents.  The sublink used here
//  is not used in the problem solution, but tracks the appropriate
//  crossovers used to produce the composite genome childern.

int
CROSSover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

  int nc=0;

//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom
    sis.head();         // set iterator to head of list
    for(int i=0; i<sis.size(); i++){
      *sis.current() = marker[i];
      sis.next();
    }
    nc += 1;            //  Return number of childern (eventually)
  }

//  "bro" is based on dad...
```

```
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);                //  Make a copy of the dad
    bro.head();         // set iterator to head of list
    for(int i=0; i<bro.size(); i++){
      *bro.current() = marker[i];
      bro.next();
    }
    nc += 1;            //  Return number of childern (eventually)
  }

  for(int j=0; j<TAGS; j++) marker[j]=0; //  Reset markers
  return nc;

}

//  Beginning of LASTover
//  LASTover returns copies of the parents.  The sublink used here
//  is not used in the problem solution, but tracks the appropriate
//  crossovers used to produce the composite genome childern.

int
LASTover(const GAGenome& p1, const GAGenome& p2,
               GAGenome* c1, GAGenome* c2){
  GAListGenome<int> &mom=(GAListGenome<int> &)p1;
  GAListGenome<int> &dad=(GAListGenome<int> &)p2;

//    Print out each genome every time LASTover is accessed
    if( dumpout == 8 ) {
      fcheck << "\n LASTover" ;
      fcheck << "\n mom:  " << mom ;   //
      fcheck << "\n dad:  " << dad ;   //
    }

  int nc=0;

//  "Sis" is based on mom...
  if(c1){
    GAListGenome<int> &sis=(GAListGenome<int> &)*c1;
    sis.GAList<int>::copy(mom);         //  Make a copy of the mom
    sis.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }

//  "bro" is based on dad...
  if(c2){
    GAListGenome<int> &bro=(GAListGenome<int> &)*c2;
    bro.GAList<int>::copy(dad);                //  Make a copy of the dad
    bro.head();         // set iterator to head of list
    nc += 1;            //  Return number of childern (eventually)
  }
  return nc;

}

//   This function cumulatively updates the crossover counters
void
sumUp(const GAGenome& p){
  GAListGenome<int> &sublist=(GAListGenome<int> &)p;

  int start = 0;

  if(pattnum < 2) { //  If only one pattern
start = 1;
```

```
  }
  if(fixed == 1){ //  If only one PCB set
  start = 2;
  }

  sublist.head();  //  Set pointer to start of list
  for(int i=start; i<TAGS; i++) {
    NN[i] = *sublist.warp(i) + NN[i]; //  Adds 1 to the appropriate
  }
}

//   This crossover function breaks up the genomes into separate npaths
//  and send each separately into appropriate crossover subfunctions.
void
tallyUpdate(const GAGenome& a){
  PanelDesignGenome& subject = (PanelDesignGenome &)a;

    sumUp(subject.path(nlists-1)); //  Trailer list is last in the genome

}

//   This function resets a trailer sublist to all zero values
void
resetTrailer(const GAGenome& p){
  GAListGenome<int> &sublist=(GAListGenome<int> &)p;

  int tempsize = sublist.size(); //  Save the size (same as TAGS)
  while(sublist.head()) sublist.destroy(); //  Get rid of values
  sublist.insert(0, GAListBASE::HEAD); //  Make a new head (=0)
  for(int i=0; i<tempsize-1; i++)
      sublist.insert(0, GAListBASE::AFTER); //All tags zero again

}

//   This function sends only the last sublist (trailer) to
//  have all its values set to zero
void
resetLast(const GAGenome& a){
  PanelDesignGenome& subject = (PanelDesignGenome &)a;

    resetTrailer(subject.path(nlists-1));//  Trailer list is last in the genome

}

//Record best genomes; lnum used to designate the bestIndividual lengths, since
//the bestIndividuals are likely shorter than ga_store lists.
void
recordList(const GAGenome& a, const GAGenome& b, int lnum){
  GAListGenome<int> &bestone=(GAListGenome<int> &)a;
  GAListGenome<int> &recorder=(GAListGenome<int> &)b;

  for(int i=0; i<listlength[lnum]; i++){
    recorder.warp(i);
    bestone.warp(i);
    *recorder.current() = *bestone.current();
  }
}

//Send lists off to be recorded, since they could be differently sized
//than the bestIndividuals
void
recordGenome(const GAGenome& a, const GAGenome& b, int rp){
  PanelDesignGenome& bestone = (PanelDesignGenome &)a;
  PanelDesignGenome& recorder = (PanelDesignGenome &)b;
```

```
  int i;

  //Record the list sizes for this particular solution
  for(i=0; i<nlists-1; i++){
    listsizes[rp][i] = listlength[i]; //Record list sizes
    //Send off lists for recording. Note that rack list could just be a dummy
    recordList(bestone.path(i), recorder.path(i), i);
  }

}

//Temporarily store the ga_store lists; necessary since the lists contain
//useful values which are less in total number than the storage list sizes
void
tempList(const GAGenome& a, int lnum, int rp){
  GAListGenome<int> &subject=(GAListGenome<int> &)a;
  int i;

  for(i=0; i<listsizes[rp][lnum]; i++){
    subject.warp(i);
    templist[lnum][i] = *subject.current();
  }
}

//Send lists off to be printed, since they could be differently sized
//than the bestIndividuals
void
tempGAstore(const GAGenome& a, int rp){
  PanelDesignGenome& subject = (PanelDesignGenome &)a;
  int i;

    //Send off the list for temporary storage to templist
  for(i=0; i<nlists-1; i++)
    tempList(subject.path(i), i, rp);

}

//  This function resets the counters NN[] and recalculates adaptive rates
void
resetTally(void){
  int cxcands = TAGS; //  Number of candidates for cx
  int start = 0;
  int i;
  int NNsum = 0; //  total number of survivors
//  over BINRESET generations
  int big = 0; //Amount which make enough survivors
  int notbig = 0; //cxcands - big
  float amt = 0.; //The float NN for each notbig
  float minperc =  //Min surv depends on popsize
float(MINSURV) / float(popsize);

  if(pattnum < 2) {
   cxcands = TAGS-1; //  If only one pattern, chop off the prob[0]
start = 1; //  value under consideration
  }
  if(fixed == 1){
cxcands = TAGS-2; //  If only one PCB set, chop off the prob[1]
  start = 2; //  value under consideration
  }

  //Check each cxcand counter for sufficient survivors

  for(i=start; i<TAGS; i++){
      if(NN[i]>=MINSURV*BINSRESET){
```

```
NNsum += NN[i]; //Find start of NNsum
        big ++; //Tally big cxcands
      }
  }

  //If none of the cxcands tallies were big enough, set to average rates
  if(big < 1) {
    for(i=start; i<TAGS; i++) prob[i] = 1. / float(cxcands);
    for(i=start; i<TAGS; i++) NN[i] = 0; //Reset tallies
    return;
  }

  notbig = cxcands - big;
  amt = minperc*(NNsum)/(1-minperc*notbig);

  for(i=start; i<TAGS; i++){
    if(NN[i]>=MINSURV*BINSRESET)
      prob[i] = float(NN[i])/( float(NNsum) + float(notbig)*amt );
    else{
      prob[i] = amt / ( float(NNsum) + float(notbig)*amt );
    }
    NN[i] = 0; //Reset counters
  }

  //  set new pcross rate, which is 1.0 minus the mutation rate
  pcross = 1.0 - prob[TAGS-1];

}

//   This crossover function breaks up the genomes into separate npaths
//  and send each separately into appropriate crossover subfunctions.
int
PanelDesignGenome::
ParseCrossover(const GAGenome& a, const GAGenome& b, GAGenome* c, GAGenome* d) {
  PanelDesignGenome& mom = (PanelDesignGenome &)a;
  PanelDesignGenome& dad = (PanelDesignGenome &)b;

  int nc=0;
  int selector = 0; //  Selector of only one action
  int start = 0;

  if(pattnum < 2) { //  If only one pattern
start = 1;
  }
  if(fixed == 1){ //  If only one PCB set
  start = 2;
  }

  if(cx < prob[0]) selector= 0;
  else if(cx < prob[0]+prob[1]) selector= 1;
  else if(cx < prob[0]+prob[1]+prob[2]) selector= 2;
  else if(cx < prob[0]+prob[1]+prob[2]+prob[3]) selector= 3;
  else if(cx < prob[0]+prob[1]+prob[2]+prob[3]+prob[4]) selector= 4;
  else selector= 4;

  if(c && d){
    PanelDesignGenome& sis = (PanelDesignGenome &)*c;
    PanelDesignGenome& bro = (PanelDesignGenome &)*d;
    for(int i=0; i<mom.npaths(); i++){
      if(i==0){
        if(pattnum>1){  //  PATTOVER can't handle one pattern
  if(selector==0){
    nc = PATTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
    marker[0] = 1;
```

```
  }
          else {
    nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
    marker[0] = 0;
  }
}
        else {
  nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
  marker[0] = 0;
}
      }
      else if(i==1){
if(selector==1){
          nc = PCBover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
  marker[1] = 1;
        }
        else {
  nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
  marker[1] = 0;
}
      }
      else if(i==2){
        switch (selector){
        case 0:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
```

```
}
      }
      else if(i==3){ //AIM Override and force to do nothing
        if(aim) selector = 0;  //for aim case; note already used selector
        switch (selector){ //for i=0, 1 and 2
        case 0:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), &bro.path(i));
break;

case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       &sis.path(i), &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
}
      }
      else if(i == (mom.npaths()-1) ){
nc = CROSSover( mom.path(i), dad.path(i),
       &sis.path(i), &bro.path(i));
      }
    }
    nc=2;
  }

  else if(c) {
    PanelDesignGenome& sis = (PanelDesignGenome &)*c;
    for(int i=0; i<mom.npaths(); i++){
      if(i==0){
        if(pattnum>1){  //  PATTOVER can't handle one pattern
  if(selector==0){
    nc = PATTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
    marker[0] = 1;
  }
         else {
    nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
    marker[0] = 0;
  }
}
```

```
        else {
  nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
  marker[0] = 0;
}
      }
      else if(i==1){
if(selector==1){
         nc = PCBover( mom.path(i),  dad.path(i), &sis.path(i), 0);
  marker[1] = 1;
        }
        else {
  nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
  marker[1] = 0;
}
      }
      else if(i==2){
        switch (selector){
        case 0:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
}
      }
      else if(i==3){ //AIM Override and force to nothing
        if(aim) selector = 0;  //for aim case; note already used selector
        switch (selector){ //for i=0, 1 and 2
        case 0:
```

```
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), &sis.path(i), 0);
break;

case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       &sis.path(i),0);
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
}
      }
      else if(i == (mom.npaths()-1) ){
nc = CROSSover( mom.path(i), dad.path(i),
       &sis.path(i), 0);
      }
    }
    nc=1;
  }

  else if(d) {
    PanelDesignGenome& bro = (PanelDesignGenome &)*d;
    for(int i=0; i<mom.npaths(); i++){
      if(i==0){
        if(pattnum>1){  //  PATTOVER can't handle one pattern
  if(selector==0){
    nc = PATTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
    marker[0] = 1;
  }
         else {
    nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
    marker[0] = 0;
  }
}
        else {
  nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
  marker[0] = 0;
}
      }
      else if(i==1){
```

```
if(selector==1){
          nc = PCBover( mom.path(i),  dad.path(i), 0, &bro.path(i));
  marker[1] = 1;
        }
        else {
  nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
  marker[1] = 0;
}
      }
      else if(i==2){
        switch (selector){
        case 0:
nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 0;
break;

case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
}
      }
      else if(i==3){ //AIM Override and force to nothing
        if(aim) selector = 0;  //for aim case; note already used selector
        switch (selector){ //for i=0, 1 and 2
        case 0:
nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
break;

        case 1:
nc = LASTover( mom.path(i),  dad.path(i), 0, &bro.path(i));
break;
```

```
case 2:
        nc = OXover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 1;
        marker[3] = 0;
        marker[4] = 0;
break;

case 3:
        nc = INVover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 1;
        marker[4] = 0;
break;

case 4:
        nc = ROTover( mom.path(i),  dad.path(i),
       0, &bro.path(i));
        marker[2] = 0;
        marker[3] = 0;
        marker[4] = 1;
break;

default:
cout<<"\n\nSomething wrong in ParseCrossover\n";
exit(1);
}
      }
      else if(i == (mom.npaths()-1) ){
nc = CROSSover( mom.path(i), dad.path(i),
       0, &bro.path(i));
      }
    }
    nc=1;
  }

  for(int j=0;j<TAGS; j++) marker[j]=0; //  Reset markers
  return nc;
}

//    A simple routine to rotate rectangles
//    Routine flips value to which pointer is pointing from 0 to 1
//    or vice-versa
//    MUST have pointer at node before call.
void
rectangle(const GAGenome& p){
  GAListGenome<int> &subject=(GAListGenome<int> &)p;
  if(*subject.current()==180) *subject.current()=0;
  else *subject.current() = 180;
}

//    A simple routine to rotate squares
//    Routine randomly selects a square position for current gene
//    (current gene is at p.current(), set before call)
void
square(const GAGenome& p){
  GAListGenome<int> &subject=(GAListGenome<int> &)p;
  int rv,degrees;
  degrees = *subject.current();
  do{
      rv = GARandomInt(0,3);
      switch (rv){
```

```
        case 0:  *subject.current() = 0;  break;
        case 1:  *subject.current() = 180;  break;
        case 2:  *subject.current() = 90;  break;
        case 3:  *subject.current() = 270;  break;
        default: cout << "\n\nrv in square() not defined!!!!!\n\n"; exit(1);
    }
  }while(degrees==*subject.current());
}

//  This is the initialization for our lists.  Create a list that is
//  nn long and whose nodes contain numbers in sequence.
//  The first thing to do in the initializer is to clear out any old
//  contents.  Notice that we have to cast the genome into the type of
//  genome we're using (in this case a list).  The GA always operates on
//  generic genomes.
//  Note that "list" below is only one string, not the multiple strings
//  of the composite genome.  Each list is sent separately.
void
PanelDesignGenome::DesignInitializer(GAGenome & c){
  GAListGenome<int> & list = (GAListGenome<int> &)c;

//  We must first destroy any pre-existing list.
  while(list.head()) list.destroy();

//  Insert the head of the list with a value of zero.

  list.insert(0, GAListBASE::HEAD);

//  Loop through listsize times and append nodes onto the end of the list.
//  The values for the PCBrotation list[1] are all zero, but other lists are
//  initialized to the index number of the list. Note that listsize is
//  changed each time Initialize loops, and DesignInitializer
//  is called for each Initialize loop.
//  Note that it does not matter what values the trailer sublist will
//  initially be set to, since it will be reset in the crossover subfunctions.

  int i;
  if(listnumber == 1){ //     This is the PCB rotations sublist
    for(i=0; i<listsize-1; i++)
      list.insert(0, GAListBASE::AFTER); //All PCB initially at zero

    for(i=0; i<listsize; i++){ //  randomize PCB rotations list
      list.next(); //    move pointer to position
      if(GARandomBit()) {
       if(sqrect[i] == 1) rectangle(list); //  Random change for rect
       if(sqrect[i] == 2) square(list); //  Random change for square
      }
    }
  }

  else if(listnumber == nlists-1){ //  The trailer sublist
    for(i=0; i<listsize-1; i++)
      list.insert(0, GAListBASE::AFTER); //All tags initially zero
  }

  else if(listnumber == 3 ){ //AIM list 3 (possibly not used)
    if(!aim){ //AIM If aim, only single 0 anyway
      for(i=0; i<listsize-1; i++)
        list.insert(i+1, GAListBASE::AFTER);
      // Now randomize the contents of the list.
      for(i=0; i<listsize; i++)
        if(GARandomBit()) list.swap(i, GARandomInt(0, listsize-1));
    }
  } //     End of "else" for the sequence lists
```

```
  else if(listnumber == 2){ //  For sequence sublist
    for(i=0; i<listsize-1; i++)
      list.insert(i+1, GAListBASE::AFTER);
    // Now randomize the contents of the list.
    for(i=0; i<listsize; i++)
      if(GARandomBit()) list.swap(i, GARandomInt(0, listsize-1));
  } //     End of "else" for the sequence lists

  else if(listnumber == 0){ //  The pattern list
      *list.current() = GARandomInt(0, pattnum-1);//  Randomize the pattern
  }
}

//  Below tells class how to scroll through the composite genome during a
//  write command.
//  The *values of the lists are returned, rather than the pointers
//  themselves. Recall that these lists are pointers, not values.
int
GAListGenome<int>::write(ostream & os) const {
  int *cur, *head;
  GAListIter<int> iter(*this);
  if((head=iter.head()) != 0) os << *head << " ";
  for(cur=iter.next(); cur && cur != head; cur=iter.next())
    os << *cur << " ";
  return os.fail() ? 1 : 0;
}

//   This subroutine reads pcb component and rack input data
void readFile( void) {
  int j;
  int fixer = 0; //A sum to check on no. of fixed PCB

  //If randomly generated component locations/types, then use testdata#.txt
  //Else, use pcbV.txt (they have different formats).
  //The filename was set before the readFile call
  if ((infile = fopen(name6,"r"))==NULL)
    {
      cout << "\nCannot open input file " << name6 << "!\n";
      exit(1);
    }

  fscanf(infile, "%d",&pcbnum);
  if(pcbnum > PCBMAX) {
      cout << "\npcbnum > PCBMAX\n";
      fclose(infile);
      exit(1);
      }

  listlength[1] = pcbnum; //Set listlength over to pcbnum

  for (j = 0; j < pcbnum; j++){
    fscanf(infile, "%d", &(sqrect[j]) );//Read shape types for each PCB
    fscanf(infile, "%f", &(PCBw[j]) );//Read PCB width
    fscanf(infile, "%f", &(PCBh[j]) );//Read PCB height
  }

  //Check width vs. ht of pcb and set pcb sqrect[]
  for (j = 0; j < pcbnum; j++){
    if(PCBw[j]==PCBh[j]) sqrect[j] = 2; //Square
    else sqrect[j] = 1; //Rectangle
  }

  //If a traditional case, then sqrect does not mattern (all fixed in
```

```
  //their pattern positions)
  if(trad){
    for (j = 0; j < pcbnum; j++) sqrect[j] = 0;
  }


  for (j=0; j<pcbnum; j++) fixer = fixer + sqrect[j];
  if (fixer<1) {  //  all pcb are fixed in space
    fixed = 1; //  so, set prob of cross on PCB to zero
    prob[1] = 0.0;
  }

  //If all locations are to be read in, then read no. of slots (aim doen't need them)
  //Else, read in ranges of components per panel, ctypes and feeder dims

  if(locgen  && aim ) {
    //Read in number of components per pcb, lo/hi
    fscanf(infile, "%d %d", &comp_low, &comp_hi);
  }

  //If the locations are to be generated, PAPM and RTHM need comp values as well
  //as ctype and slot parameter values
  if(locgen && !aim) {
    //Read in number of components per pcb, lo/hi
    fscanf(infile, "%d %d", &comp_low, &comp_hi);
    //Read in number of component types per panel, lo/hi
    fscanf(infile, "%d %d", &ctyp_low, &ctyp_hi);
    //For experiments, we will have only one side with a rack
    //We will generate the slot locations from left to right,
    //with the perspective of the rack being at the bottom of the panel
    fscanf(infile, "%f %f %f",&bottom_clear, &slotspace, &middlespace);
  }

  //Lastly, RTHM needs the following, regardless of locgen:
  //heads, index time, table velocity, and slot time
  if(rthm)fscanf(infile, "%d %f %f %f", &heads, &indextime, &tablevel, &slottime);

      fclose(infile);

}

//   Generate locations for PCB and rack
void createLocations( int kkk , int seeding ) {
  int i,j;
  int compnum=0; //Components per pcb (varies)
  int kount = 0; //Panel component counter START @ ZERO!
  int ctypesum = 0;
  int looplimit = 0;
  float slotzero = 0.; //Leftmost feeder location
  int endloop; //Sets how far to take random loc gens

  //Set a random seed; this method allows repeated runs to have the same xy
  //yet replications within that run to have different xy

  GARandomSeed((unsigned int)seeding);

  //Set number of types for the panel
  if(aim) racknum = 1;
  else racknum  = GARandomInt(ctyp_low, ctyp_hi);

  listlength[3] = racknum; //Set listlength over to racknum

  for(i=0; i<repeats; i++)  //Store for results output
    results[kkk+i].paneltypes = racknum;//Note that the value is same for
```

```
//next three "repeats"

  int ctype_check[TOTALSLOTMAX]; //in IRIX, size must be static
  for(i=0; i<racknum; i++) ctype_check[i] = 0;

// The Alike aspect can be incorporated right here.
// Set the endloop = pcbnum if not alike (all different), and
// endloop = 1 if all alike.  The 1 will force the loop to run only once.
  if(alike) endloop = 1;
  else      endloop = pcbnum;

  for (j = 0; j < endloop; j++){

    //Generate number of components and types for first pcb

    compnum = GARandomInt(comp_low, comp_hi);

    for(i=0; i<repeats; i++)  //Store for results output
      results[kkk+i].pcbcomp[j] = compnum;//Note that the value is same for
  //next "repeats"

    //Recall that xloc and yloc are relative to pcb centers
    for (i = 0; i < compnum; i++) {
      (comp[kount]).xloc = GARandomFloat( -(PCBw[j])/2., (PCBw[j])/2. );
      (comp[kount]).yloc = GARandomFloat( -(PCBh[j])/2., (PCBh[j])/2. );
      (comp[kount]).ct   = GARandomInt( 0, racknum-1 );
      ctype_check[ (comp[kount]).ct ] = 1; // A check on types
      (comp[kount]).pcbid = j;
      kount++;
    }
  }


  //Need to check if all component types were used; if not, randomly select
  //a few components and change ctype, then check again.
  //Likely will select 2 components per pass in the pcb

  //Another compensation for alike: Use endloop again, but this time
  //it represents the number of times the check is performed.
  //Restricts to one pcb's components if all pcb are alike, but
  //will allow all panel to be searched if pcb are all different
  if(!aim){
  if(alike) endloop = compnum;
  else      endloop = kount;

  for(i=0; i<racknum; i++) ctypesum += ctype_check[i]; //cypesum ?= racknum

  if(ctypesum<racknum) {
    do{
      for(i=0; i<racknum; i++) ctype_check[i] = 0;
//      for(j=0; j<listlength[2]; j++){
      for(j=0; j<kount; j++){
        if(GAFlipCoin(2./float(compnum)))  //Keep old ct if FALSE
           (comp[j]).ct   = GARandomInt( 0, racknum-1 ); //Set a new ct
      ctype_check[ (comp[j]).ct ] = 1;
      }
      looplimit++;
      for(i=0; i<racknum; i++) ctypesum += ctype_check[i];
      if(looplimit>100){
        cout<<"\n\nERROR IN ctypesum= "<<ctypesum<<"\tracknum = "<<racknum<<"\n";
        exit(1);
      }
    }while(ctypesum<racknum);
  }
```

```
  }

    // Another 'alike' compensation:  this is not run if all pcb different;
    // i.e., alike = 0.
    // If all the PCB are like, just copy subsequent component locations
    // and types correspondingly; ups the kount, thus setting total
    //component count for panel
    if(alike){
      for(i=0; i<repeats; i++)  //Store for results output
        results[kkk+i].pcbcomp[j] = compnum;//Note that the value is same for
  //next "repeats"
      for (j = 0; j < pcbnum-1; j++){
        for (i = 0; i < compnum; i++) {
          (comp[kount]).xloc = (comp[i]).xloc;
          (comp[kount]).yloc = (comp[i]).yloc;
          (comp[kount]).ct   = (comp[i]).ct;
          ctype_check[ (comp[kount]).ct ] = 1; // A check on types
          (comp[kount]).pcbid = j+1;
          kount++;
        }
      }
    }

  listlength[2] = kount; //Set listlength to TOTAL number of components

  for(i=0; i<repeats; i++)  //Store for results output
    results[kkk+i].panelcomps = kount;//Note that the value is same for
  //next "repeats"

  //NEXT:  SLOT LOCATIONS.  Assumes one bank at bottom of panel
  //Find left starting slot location
  if(!aim){
    slotzero =  middlespace - slotspace*float(racknum)/2.;

    for (i = 0; i < racknum; i++) {
      (slot[i]).xglob = slotzero + float(i)*slotspace;
      (slot[i]).yglob = - bottom_clear;
      (slot[i]).ct = i; // Just for initialization
    }
  }
}

//   Read locations for PCB and rack from input file
void readLocations( int kkk ) {
  int i,j;
  int compnum; //Components per pcb (varies)
//  int rackcheck = 0; //A check on separate file inputs
  int kount = 0; //Panel component counter START @ ZERO!

  if ((infile = fopen(name8,"r"))==NULL)
    {
      cout << "\nCannot open input file " << name8 <<"!\n";
      exit(1);
    }

  for(i=0; i<repeats; i++)  //Store for results output
    results[kkk+i].paneltypes = racknum;//Note that the value is same for
//next three "repeats"

  for (j = 0; j < pcbnum; j++){
    fscanf(infile, "%d",&compnum); //Read number of components for PCB

    for(i=0; i<repeats; i++)  //Store for results output
      results[kkk+i].pcbcomp[j] = compnum;//Note that the value is same for
```

```
  //next three "repeats"

    if(kount+compnum > TOTALCOMPMAX) { // Look ahead to see if max comps exceeded
       cout << "\nkount > TOTALCOMPMAX\n";
       fclose(infile);
       exit(1);
    }
    for (i = 0; i < compnum; i++) {
      fscanf(infile, "%f",&(comp[kount]).xloc);
      fscanf(infile, "%f",&(comp[kount]).yloc);
      if(!aim) fscanf(infile, "%d",&(comp[kount]).ct);
      else (comp[kount]).ct = 0;
      (comp[kount]).pcbid = j;
      kount++;
    }

  } //End of file reading

  listlength[2] = kount; //Set listlength to TOTAL number of comps

  for(i=0; i<repeats; i++)  //Store for results output
    results[kkk+i].panelcomps = kount;//Note that the value is same for
  //next three "repeats"

  //NEXT:  RACK DATA (simply not done for AIM case, slot[] left uninitialized
    if(!aim){
      fscanf(infile, "%d",&racknum); //Read number of components for PCB
      for (i = 0; i < racknum; i++) {
        fscanf(infile, "%f",&(slot[i]).xglob);
        fscanf(infile, "%f",&(slot[i]).yglob);
        (slot[i]).ct = i; // Just for initialization
      }
    }

  if(!aim) listlength[3] = racknum; //Set listlength over to racknum
  else     listlength[3] = 1; //Set to a value not used

  //NOTE:  Need to calculate the slotspace (i.e., distance between slots)
  //Assumes that there is the same distance between any and all slots
  slotspace = fabs(slot[0].xglob - slot[1].xglob);

  fclose(infile);

}

//   This subroutine reads pattern input data
void readFile3(void) {
  int i,j,pattcount;
  float probsum = 0.;

    if ((infile = fopen(name7,"r"))==NULL)
        {
          cout << "\nCannot open " << name7 << " !\n";
          exit(1);
        }

  fscanf(infile, "%d",&pattnum);
  if(pattnum > PATTERNMAX) { //Check on maximum no. of patterns
      cout << "\npattnum > Maximum number of Patterns\n";
      fclose(infile);
      exit(1);
      }

  if(pattnum < 2) prob[0] = 0.0; //  If only one pattern, then
```

```
//  prob of changing patterns is zero

  listlength[0] = 1; //  Set listlength to 1, regardless

  fscanf(infile, "%d",&pcbnum);
  if(pcbnum != listlength[1]) { //Check that pcbnum in patterns.txt matches
//pcbnum in pcbV.txt
  cout << "\npcbnum from testdata.txt(or pcbdata) != pcbnum from patterns.txt \n";
    fclose(infile);
    exit(1);
  }

  for (j = 0; j < pattnum; j++){
    fscanf(infile, "%d",&pattcount); //Read id of pattern (a dummy)
//pattcount is just in file for readability
    for (i = 0; i < pcbnum; i++) {
      fscanf(infile, "%f",&(patt[j]).XX[i]);
      fscanf(infile, "%f",&(patt[j]).YY[i]);
      fscanf(infile, "%d",&(patt[j]).OO[i]);
    }
  } //End of file reading

  fclose(infile);

  //  Check for correct probability input
  if(!probcheck){
    if(!probcheck) probcheck = 1; //This is for args check, but only done
//after pattnum is read once
    if(pattnum > 1 ){ //  Most general case; have not
      for(i=0; i<TAGS; i++){ //  accounted for all PCB non-rotating
        prob[i] = prob[i]/100.; //  Converts percentages to decimals
        prob_save[i] = prob[i]; //Save original values
        probsum = probsum + prob[i];
      }
    }

    if(pattnum < 2 && fixed == 1){ //  Only one pattern and all
      prob[0] = prob[1] = 0.; //  PCB are fixed
      for(i=2; i<TAGS; i++){
        prob[i] = prob[i]/100.;
        prob_save[i] = prob[i]; //Save original values
        probsum = probsum + prob[i];
      }
    }
    else
    if(pattnum < 2 ){ //  Only one pattern
      prob[0] = 0.; //  but PCB still can rotate
      for(i=1; i<TAGS; i++){
        prob[i] = prob[i]/100.;
        prob_save[i] = prob[i]; //Save original values
        probsum = probsum + prob[i];
      }
    }
    if(probsum < .999 ){
      cout << "\nSum of probabilities is less than 0.999\n";
      cout << "For " << pattnum << " patterns and fixed = " << fixed << "\n";
      for(i=0; i<TAGS; i++){
        cout << "\nprob[" << i << "] = " << prob[i];
      }
      cout << "\n";
      exit(1);
    }

    if(probsum > 1.001){
```

```
      cout << "\nSum of probabilities exceeds 1.001\n";
      cout << "For " << pattnum << " patterns and fixed = " << fixed << "\n";
      for(i=0; i<TAGS; i++){
        cout << "\nprob[" << i << "] = " << prob[i];
      }
      cout << "\n";
      exit(1);
    }

  }
  //  new pcross is 1.0 minus the mutation rate
  pcross = 1.0 - prob[TAGS-1];

  //Append the new information to the locations file
  if(locgen){
    flocs.open(name5, ios::out | ios::app );
    flocs << "\n";
    for (i=0; i<pcbnum; i++)  flocs << sqrect[i]
      << " " << patt[0].XX[i] << " " << patt[0].YY[i]
        << " " << patt[0].OO[i] << "\n";
    flocs << "\n";
    flocs.flush();
    flocs.close();
  }

}

/*************************************************************************/
// Assigns index of panel output file name as a character
// Used for multiple output of ACAD script files
/*************************************************************************/
char *index1( int j ) {
/* Up to INDEXMAX panel output files are writable */
/* (Included 20 characters for possible expansion */
static char *textnum[]=
{
  "0",
  "1",
  "2",
  "3",
  "4",
  "5",
  "6",
  "7",
  "8",
  "9",
  "10",
  "11",
  "12",
  "13",
  "14",
  "15",
  "16",
  "17",
  "18",
  "19",
  "X"
};

if (j >= 0 && j <= 20 )
return (textnum[j]);
else
{
printf("\nNumber of index exceeds %d", 20);
```

```
return (textnum[0]);
exit(1);
}
}

/*************************************************************
// Write a solution to ACAD script file
// Acad Script writing.  Writes a script file for Autocad
// to show the sequence of placements for the given solution
*************************************************************/
void WriteAcadSeq( const GAGenome& p){
  PanelDesignGenome & gen=(PanelDesignGenome &)p;
float score = 0;
int i,j,k;
float offset = 325.0; //Offset between views

fcheck << "\nWriting ACAD file for \n";
fcheck << gen << "\n";

//Print out the acad script file for reference in documentation
acadout <<  "TEXT\n";
        acadout <<  0 << "," << 305.0 << "\n6.0\n0\n";
acadout <<  acadname << "\n";

if(trad){
  for (i=0; i<listlength[3]; i++){
    assign[*gen.path(3).warp(i)] = i;
  }
  //sx, sy are never used in trad cases, but are handy here for plotting
  //Since they were not calculated in sx, sy, they are calculated now
  //Reading left to right:
  //Value for component type j in a sublist position is assigned
  //to slot position in order.
  //SO, think like this:
  // sx[component type at sublist position] = slot[sublist postion].x/y
  if(!aim){
    for (j=0; j<listlength[3]; j++){
      sx[*gen.path(3).warp(j)] = slot[j].xglob;
      sy[*gen.path(3).warp(j)] = slot[j].yglob;
     }
}

}

//Need to recalculate x,y,sx,sy for panelized situation
//Otherwise, the x,y are set, fixed by setupMatrix
        if(!trad) {
  DistanceCalc(gen); //Calculate x,y,sx,sy arrays

          if(aim) score = DistAIM( ); //Distance for AIM problem
          if(papm) score = DistPAPM(gen); //Distance for the PAPM problem.
          if(rthm) score = DistRTHM(gen); //Distance for the RTHM problem.
}
//We could have gotten score from score_list either way, actually...
else score = score_list[best_index];

//  Set linetype to dashed for PCB borders
acadout  << "LINETYPE\nSET\nDASHED\n\n";

//  Draw rectangles for the PCB in this solution
for( k = 0; k<listlength[1]; k++){

  acadout <<  "RECTANG\n";
  acadout  << -PCBw[k]/2.0 << "," << -PCBh[k]/2.0 << "\n";
```

```
  acadout  << PCBw[k]/2.0 << "," << PCBh[k]/2.0 << "\n";

    //  If a rectangle and .OO = 90 degrees, rotate boundary
    if(sqrect[k] == 1 && patt[*gen.path(0).warp(0)].OO[k] == 90 ){
      acadout << "ROTATE\nLAST\n\n" << "0.0,0.0,0.0\n90\n";
    }

  //  Move PCB to correct XX and YY position
  acadout << "MOVE\nLAST\n\n" << "0.0,0.0,0.0\n";
  acadout << patt[*gen.path(0).warp(0)].XX[k] << ",";
  acadout << patt[*gen.path(0).warp(0)].YY[k] << "\n";
}

//  Set linetype back to continous for rest of work
acadout  << "LINETYPE\nSET\nCONTINUOUS\n\n";

//  Print a circle for the starting slot
//This works for either trad or !trad
        if(!aim){
    acadout <<  "CIRCLE\n";
    acadout  << sx[comp[*gen.path(2).warp(0)].ct] << ",";
    acadout  << sy[comp[*gen.path(2).warp(0)].ct] ;
    acadout <<  "\n" << 5.0 << "\n";
        }

//  Then a circle for first component placement
if(!trad){
  acadout <<  "CIRCLE\n";
  acadout  << x[0];
  acadout  << "," <<  y[0];
  acadout <<  "\n" << 2.0 << "\n";
        }
else{
  acadout <<  "CIRCLE\n";
  acadout  << x[*gen.path(2).warp(0)];
  acadout  << "," <<  y[*gen.path(2).warp(0)];
  acadout <<  "\n" << 2.0 << "\n";
        }
/**********AIM requires only moves from component x[i],y[i] to *************
***********component x[i+1],y[i+1] or equivalent*********************************/

if(aim || rthm){
        //  Print component positions
          for (i=0; i<listlength[2]; i++){
  if(!trad){
    if(rect){
              acadout <<  "RECTANGLE\n";
              acadout << x[i]-1
      << "," << y[i]-1 << "\n";
              acadout << x[i]+1
      << "," << y[i]+1 << "\n";
    }
    else{
              acadout <<  "TEXT\n";
              acadout << x[i]  << "," << y[i] ;
              acadout << "\n6.0\n0\n";
              acadout <<  i << "\n";
    }
          }
  else{
    if(rect){
              acadout <<  "RECTANGLE\n";
              acadout << x[*gen.path(2).warp(i)]-1
      << "," << y[*gen.path(2).warp(i)]-1 << "\n";
```

```
            acadout << x[*gen.path(2).warp(i)]+1
      << "," << y[*gen.path(2).warp(i)]+1 << "\n";
    }
    else{
            acadout <<  "TEXT\n";
            acadout << x[*gen.path(2).warp(i)]  << "," << y[*gen.path(2).warp(i)] ;
            acadout << "\n6.0\n0\n";
            acadout <<  *gen.path(2).warp(i) << "\n";
    }
        }
        }
        //Print a continuous polyline of all moves
        acadout <<  "PLINE\n";
        for (i=1; i<listlength[2]; i++){
    if(!trad){
            acadout  << x[i-1] << ",";
            acadout  << y[i-1] << "\n";
            acadout  << x[i]  << ",";
            acadout  << y[i] << "\n";
          }
    else{
            acadout  << x[*gen.path(2).warp(i-1)] << ",";
            acadout  << y[*gen.path(2).warp(i-1)] << "\n";
            acadout  << x[*gen.path(2).warp(i)]  << ",";
            acadout  << y[*gen.path(2).warp(i)] << "\n";
          }
        }
  //Close the path:
  if(!trad){
          acadout  << x[0]  << ",";
          acadout  << y[0] << "\n\n";
        }
  else{
    acadout  << x[*gen.path(2).warp(0)] << ","
      <<  y[*gen.path(2).warp(0)] <<  "\n\n";
  }
}
/**********PAPM:  For clarity, print "required moves" and non-required moves
**********in two separate views************************************/
else if(papm){
  //  duplicate all drawn thus far and move copy to new location
  //  Now copy and duplicate boundaries, while no lines are yet drawn
  acadout << "COPY\nALL\n\n" << "0.0,0.0,0.0\n";
  acadout << offset << ",0.0\n";

  //  Print legends over the two views
    acadout <<  "TEXT\n";
    acadout <<  offset << ",350" << "\n6.0\n0\n";
    acadout <<  "NON-REQUIRED" << "\n";
    acadout <<  "TEXT\n";
    acadout <<  "0.0,350" << "\n6.0\n0\n";
    acadout <<  "REQUIRED" << "\n";

  //  Print component positions
        for (i=0; i<listlength[2]; i++){
    if(!trad){
      if(rect){
                acadout <<  "RECTANGLE\n";
                acadout << x[i]-1  << "," << y[i]-1 << "\n";
                acadout << x[i]+1  << "," << y[i]+1 << "\n";
      }
      else{
        acadout <<  "TEXT\n";
        acadout << x[i]  << "," << y[i] ;
```

```
        acadout << "\n6.0\n0\n";
        acadout <<  i << "\n";
      }
           }
    else{
      if(rect){
                acadout <<  "RECTANGLE\n";
                acadout << x[*gen.path(2).warp(i)]-1
<< "," << y[*gen.path(2).warp(i)]-1 << "\n";
                acadout << x[*gen.path(2).warp(i)]+1  << ","
<< y[*gen.path(2).warp(i)]+1 << "\n";
      }
      else{
        acadout <<  "TEXT\n";
        acadout << x[*gen.path(2).warp(i)]  << "," << y[*gen.path(2).warp(i)] ;
        acadout << "\n6.0\n0\n";
        acadout <<  *gen.path(2).warp(i) << "\n";
      }
    }
         }

  // Print LINE and vertices for "non-required" moves
  //noting that sx,sy are o.k for either trad or !trad
  for (i=0; i<listlength[2]-1;  i++) {
  acadout <<  "LINE\n";
    if(!trad){
      acadout  << x[i]+ offset << ",";
      acadout  << y[i] << "\n";
    }
    else{
      acadout  << x[*gen.path(2).warp(i)]+ offset << ",";
      acadout  << y[*gen.path(2).warp(i)] << "\n";
    }
    acadout  << sx[comp[*gen.path(2).warp(i+1)].ct]+ offset << ",";
    acadout  << sy[comp[*gen.path(2).warp(i+1)].ct] << "\n\n";
  }

  // Print LINE and vertices for "required" moves
  for (i=0; i<listlength[2];  i++) {
  acadout <<  "LINE\n";
    acadout  << sx[comp[*gen.path(2).warp(i)].ct] << ",";
    acadout  << sy[comp[*gen.path(2).warp(i)].ct] << "\n";
    if(!trad){
      acadout  << x[i] << ",";
      acadout  << y[i] << "\n\n";
    }
    else{
      acadout  << x[*gen.path(2).warp(i)] << ",";
      acadout  << y[*gen.path(2).warp(i)] << "\n\n";
    }
  }

         //Print a continuous polyline of all moves, so that measurement is easier
        acadout <<  "PLINE\n";
  if(!trad){
          for (i=0; i<listlength[2]; i++){
      acadout  << sx[comp[*gen.path(2).warp(i)].ct] + 2.0*offset << ",";
      acadout  << sy[comp[*gen.path(2).warp(i)].ct] << "\n";
      acadout  << x[i] + 2.0*offset << ",";
      acadout  << y[i] << "\n";

          }
        }
  else {
```

```
    for (i=0; i<listlength[2]; i++){
      acadout  << sx[comp[*gen.path(2).warp(i)].ct] + 2.0*offset << ",";
      acadout  << sy[comp[*gen.path(2).warp(i)].ct] << "\n";
      acadout  << x[*gen.path(2).warp(i)] + 2.0*offset << ",";
      acadout  << y[*gen.path(2).warp(i)] << "\n";
    }
          }
  acadout  << "\n";
  }

//Slot print-outs are options for either rthm or papm
//Again, sx,sy work for trad or !trad
if(rthm || papm){
    for (i=0; i<listlength[3]; i++){
      acadout << "TEXT\n";
      acadout  << sx[comp[*gen.path(2).warp(i)].ct] << ",";
      acadout  << sy[comp[*gen.path(2).warp(i)].ct]-1
      << "\n6\n0\n";
      acadout << comp[*gen.path(2).warp(i)].ct << "\n";
          }
        }

//Print out score
acadout <<  "TEXT\n";
if(papm) acadout <<  2.0*offset << "," << -5.0 << "\n6.0\n0\n";
        else acadout <<  0 << "," << -5.0 << "\n6.0\n0\n";
acadout <<  "SCORE=" << score / float(listlength[2])<< "  (per component)\n";

/* Print zoom commands to finish */
acadout <<  "ZOOM\nALL\n";
acadout <<  "ZOOM\n.8X\n";
}


int
main(int argc, char *argv[])
{
  int i, ii, j, k, kk, kkk;
  int gennum = 50; //  A default termination number
  int offspring = 10; //  A default number to replace each gen
  float pconv  = 1.00001;       //  % of which we look for convergence
  int nconv    = 100;           //nconv gens back for which to find pconv
  int convg    = 1;             //Flag on whether to convg (1=yes) or not (=0)
  int starttime = time(0); //Used for absolute timing of program
  time_t timer; //Used to catch time for files writing
  char * timepoint; //Used to catch time for files writing
  int seed = 1; //A local seed storage variable
  int seeder[MAXREPLC];    // storage of random seeds for replc
  int inc_replc = 0; //Increment used to track replc

  time( &timer);
  timepoint = ctime( &timer ); //A time hack, used for file management
  //Replace the blanks in the time hack with "_"
  k=0;
  do{
  k++;
  if(*(timepoint+k)==' ') *(timepoint+k) = '_';
  }while(*(timepoint+k)!='\0' && k<30);
  cout << timepoint <<"\n";

  cout.flush();

// Parse the command line for arguments.
//   Take care here, since I want to keep this "hardwired" eventually.
```

```
  for(i=1; i<argc; i++){
    if(strcmp("popsize", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": number in population needs a value.\n";
        exit(1);
      }
      else{
popsize = atoi(argv[i]);
        continue;
      }
    }

    if(strcmp("convg", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": convg needs a value.\n";
        exit(1);
      }
      else{
convg = atoi(argv[i]);
        continue;
      }
    }

    if(strcmp("pconv", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": Percent of convergence needs a value.\n";
        exit(1);
      }
      else{
pconv = atoi(argv[i]);
        continue;
      }
    }

    if(strcmp("nconv", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": number of generations to converge needs a value.\n";
        exit(1);
      }
      else{
nconv = atoi(argv[i]);
        continue;
      }
    }

    if(strcmp("ngen", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": number of generations needs a value.\n";
        exit(1);
      }
      else{
gennum = atoi(argv[i]);
        continue;
      }
    }
    if(strcmp("mprob", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": mprob needs a value.\n";
        exit(1);
      }
      else{
mprob = float(atof(argv[i]))/100.;
        continue;
```

```
      }
    }

    if(strcmp("prob0", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob0 needs a value.\n";
        exit(1);
      }
      else{
prob[0] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("prob1", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob1 needs a value.\n";
        exit(1);
      }
      else{
prob[1] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("prob2", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob2 needs a value.\n";
        exit(1);
      }
      else{
prob[2] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("prob3", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob3 needs a value.\n";
        exit(1);
      }
      else{
prob[3] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("prob4", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob4 needs a value.\n";
        exit(1);
      }
      else{
prob[4] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("prob5", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": prob5 needs a value.\n";
        exit(1);
      }
      else{
```

```
prob[5] = float(atoi(argv[i]));
        continue;
      }
    }

    if(strcmp("dumpout", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": dumpout needs a value.\n";
        exit(1);
      }
      else{
dumpout = atoi(argv[i]);
        continue;
      }
    }

    //Number of repeats per replication
    else if(strcmp("repeats", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": repeats needs a value\n";
        exit(1);
      }
      else{
repeats = atoi(argv[i]);
        continue;
      }
    }

    //Number of replications to do for this case
    else if(strcmp("replc", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": replc needs a value\n";
        exit(1);
      }
      else{
replc = atoi(argv[i]);
        continue;
      }
    }

    //check identical pcb for random location generation
    else if(strcmp("alike", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": alike needs a value\n";
        exit(1);
      }
      else{
        alike = atoi(argv[i]);
        continue;
      }
    }

    //check for panelization (to know which file to read)
    else if(strcmp("trad", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": trad needs a value\n";
        exit(1);
      }
      else{
trad = atoi(argv[i]);
        continue;
      }
    }
```

```
    //Generate xy (and ctype) locations randomly
    else if(strcmp("locgen", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": locgen needs a value\n";
        exit(1);
      }
      else{
locgen = atoi(argv[i]);
        continue;
      }
    }

    //rect tells WriteAcad whether to print component numbers or small rectangles
    else if(strcmp("rect", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": rect needs a value\n";
        exit(1);
      }
      else{
rect = atoi(argv[i]);
        continue;
      }
    }

    //number of pcb (necessary only if multiple runs in same directory)
    else if(strcmp("pcbb", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": pcbb needs a value\n";
        exit(1);
      }
      else{
pcbb = atoi(argv[i]);
        continue;
      }
    }

    //designate an input file designator
    else if(strcmp("innum", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": innum needs a value\n";
        exit(1);
      }
      else{
innum = atoi(argv[i]);
        continue;
      }
    }

    //designate an input file designator
    else if(strcmp("aim", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": aim needs a value\n";
        exit(1);
      }
      else{
aim =  1;
papm = 0;
rthm = 0;
        continue;
      }
    }

    //designate an input file designator
    else if(strcmp("papm", argv[i]) == 0){
```

```
      if(++i >= argc){
        cerr << argv[0] << ": papm needs a value\n";
        exit(1);
      }
      else{
papm = 1;
aim =  0;
rthm = 0;
        continue;
      }
    }

    //designate an input file designator
    else if(strcmp("rthm", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": rthm needs a value\n";
        exit(1);
      }
      else{
rthm = 1;
aim =  0;
papm = 0;
        continue;
      }
    }

    else if(strcmp("seed", argv[i]) == 0){
      if(++i < argc) continue;
      continue;
    }

    else {
      cerr << argv[0] << ":  unrecognized argument: " << argv[i] << "\n\n";
      exit(1);
    }
  }

  //Tack on a designator for machine type to time hack
  //Note that this k value is from way up there, from timepoint
  //replacements of spaces...
  if(aim) *(timepoint+k-1) = 'A';
  else if(papm) *(timepoint+k-1) = 'P';
  else if(rthm) *(timepoint+k-1) = 'R';

  for(i = 0; i<TAGS; i++) marker[i] = 0; //  Initialize tags
  for(i = 0; i<repeats*replc; i++) score_list[i] = 1; //Initialize score record

  //  Number of children to generate.  Note that this DOES NOT mean that
  //  this value is the number of children to survive to next generation.
  //  That is due to the custom step and cull functions.  Still, must
  //  keep total number of tmpPop to less than popsize in order to use
  //  a lot of the standard objects in the galib.

  if(popsize % 2 == 0)  //  Must know how many to replace
    offspring = popsize- 2; //  Even population
  else
    offspring = popsize - 1; //  Odd population

  //Set the input file name before the call to readFile
  if(aim) strcpy(name6, "aim");
  else if(papm) strcpy(name6, "papm");
  else if(rthm) strcpy(name6, "rthm");
  if(locgen) {
    strcat(name6, inputf );
```

```
  strcat(name6, index1(pcbb)); //Result is either testdata#.txt
  strcat(name6, ".txt");
}
else if(pcbb>0) {
  strcat(name6, inputf2 );
  strcat(name6, index1(pcbb));
  strcat(name6, ".txt"); //or pcbV#.txt
}
else {
  cout << "\npcbb = 0\n" ;
  exit(1);
}

//Set the input pattern file name before the call to readFile
if(trad ) strcpy(name7, input2f ); //pattV#_#
else if(!trad) strcpy(name7, input2f2 ); //patterns#
strcat(name7, index1(pcbb));
if(trad) strcat(name7, index1(innum)); //the second # in pattV#_#.txt
strcat(name7, ".txt");

//Set the input locations file name (may not be used, however)
if(aim && !innum) {
  strcpy(name8, "aim" );
  strcat(name8, locations);
  strcat(name8, index1(innum));
  strcat(name8, ".txt");
}
else if(papm && !innum) {
  strcpy(name8, "papm" );
  strcat(name8, locations);
  strcat(name8, index1(innum));
  strcat(name8, ".txt");
}
else if(rthm && !innum) {
  strcpy(name8, "rthm" );
  strcat(name8, locations);
  strcat(name8, index1(innum));
  strcat(name8, ".txt");
}
else if(aim && innum > 0){
   strcpy(name8, "dA" );
   strcat(name8, index1(innum) );
   strcat(name8, ".txt" );
}
else if( (papm || rthm) && innum > 0){ //PAPM and RTHM require ctypes
   strcpy(name8, "dP" );
   strcat(name8, index1(innum) );
   strcat(name8, ".txt" );
}
readFile(); //Read parameters of PCBs and feeders

strcpy(name5, locs ); // A random generation case
if(locgen==0 && trad==0) strcat(name5, "FP");
else if(locgen==0 && trad==1) strcat(name5, "FV");
else if(locgen==1 && trad==0) strcat(name5, "RP");
else if(locgen==1 && trad==1) strcat(name5, "RV");

 strcat(name5, index1(pcbnum)); // Attach the number of pcb to filename
 strcat(name5, timepoint); // Attach a time hack
 strcat(name5, ".dat" ); // Add filename extension.

 strcpy(name4, check); // Set the filename base text
 strcat(name4, timepoint); // Attach the start time to filename
 strcat(name4, ".dat" ); // Add filename extension.
```

```
   strcpy(name1, acad_base); // Set the filename base text
   strcat(name1, timepoint); // Attach the start time to filename

   if(locgen==0 && trad==0 && !innum) strcpy(name0, "caseFP");
     else if(locgen==0 && trad==1 && !innum) strcpy(name0, "caseFV");
     else if(locgen==1 && trad==0 && !innum) strcpy(name0, "caseRP");
     else if(locgen==1 && trad==1 && !innum) strcpy(name0, "caseRV");
     else if(!locgen && trad==0 && innum>0) strcpy(name0, "dp");
     else if(!locgen && trad==1 && innum>0) strcpy(name0, "dt");
     else {
       cout << "\nError: trad, innum, locgen are: "
            << trad << ", " << innum << ", " << locgen << ",\n";
       exit(1);
     }
     strcat(name0, index1(pcbnum)); // Attach the number of pcb to filename
     strcat(name0, timepoint); // Attach a time hack
     strcat(name0, ".dat" ); // Add filename extension.

     fcheck.open(name4); //  Open a check log file

  // The next set of commands set up the genetic algorithm parameters
  // You don't need the xy information for this, just the correct
  // number of sublists (always = 5 for PAPM)
  // These commands DO NOT INITIALIZE the ga yet
  PanelDesignGenome genome(nlists);
  CarefulMatingGA ga(genome);
  ga.minimize(); //
  ga.nGenerations(gennum);
  ga.populationSize(popsize);
  GALinearScaling lin;
  ga.scaling(lin);
  ga.nConvergence(nconv);
  ga.pConvergence(pconv);
  ga.pCrossover(pcross);
  ga.nReplacement(offspring);         // number of individuals to replace each gen

  //These commands simply create a "spare" population to hold the
  //different GA candidates produced during the replications.
  //No GA evolution is performed upon them.

  CarefulMatingGA ga_store(genome); //Name of storage genome population
  ga_store.populationSize(repeats*replc);//Need number of replications*repeats
  listlength[2] = TOTALCOMPMAX; //Set to max length for storage
  listlength[3] = TOTALSLOTMAX; //Set to max length for storage
  ga_store.initialize(); //Initialize lists (recall that these
//are initialized to random values

  fcheck << "#"; //Allow for gnuplot "comments"
  fcheck << name4 <<"\n" << "#";  //Print out time hack
  for(i=1; i<argc; i++) fcheck << argv[i] << " "; //Print the command args
  fcheck << "\n";

  //Start a locations (and types) record file for the random generations
  if(locgen){
    flocs.open(name5, ios::out | ios::app );//Open or append a locations file
    flocs << replc << "\n" << repeats << "\n" << pcbnum << "\n";
    flocs.flush();
    flocs.close();
  }

// See if we've been given a seed to use (for testing purposes).  When you
// specify a random seed, the evolution will be exactly the same each time
// you use that seed number.
```

```
  for( ii=1; ii<argc; ii++) {
    if(strcmp(argv[ii++],"seed") == 0) {
      GARandomSeed((unsigned int)atoi(argv[ii]));
      seed = atoi(argv[ii]);
    }
  }

  //Once the random seed was initiated, we can
  //set up a string of random seeds for generating xy data
  //This is not necessary for locations read in from files
  if(locgen == 1)
    for(i=0; i<replc; i++)
      seeder[i] = (i+1)*seed;

  int dfile = innum; //A trace on dfile under consideration

  //START OF REPLICATIONS
  for(kkk=0; kkk<repeats*replc; kkk++){

  //Below is the only set of commands which either read in the component/
  //feeder locations or create them every new repeats set
  //Note that this is necessary BEFORE the repeats GA begin
  if(locgen  && (!kkk || (kkk % repeats)==0 ) ) {
    if(!kkk) readFile3(); //Read pattern set (only once)
    createLocations( kkk, seeder[inc_replc] ); //Generated locations and types
    inc_replc++;
  }
  //just read once if innum ==0 (this is a <mach>locations0.txt file)
  else if(!locgen && !innum && !kkk) {
        readFile3(); //Read pattern set
readLocations(kkk);
   if(trad) setupMatrix();
  }
  //Here is the trick for automatically running sucessive, deterministic cases.
  //innum > 0 starts the process, since it did not do it above
  else if(!locgen && (!kkk || ((kkk % repeats)==0 ) )) {
    readFile3(); //Read pattern set
    readLocations(kkk); //Do multiple reads
    if(trad) setupMatrix(); //Set up the matrix again
    dfile++; //increment dfile  and
    if(aim && innum > 0){ //set for next read
      strcpy(name8, "dA" );
      strcat(name8, index1(dfile) );
      strcat(name8, ".txt" );
    }
    else if( (papm || rthm) && innum > 0){ //PAPM and RTHM require ctypes
      strcpy(name8, "dP" );
      strcat(name8, index1(dfile) );
      strcat(name8, ".txt" );
    }
    if(trad){
      strcpy(name7, input2f ); //pattV#_#
      strcat(name7, index1(pcbb));
      strcat(name7, index1(dfile)); //the second # in pattV#_#.txt
      strcat(name7, ".txt");
    }
  }

//This is the initialization of a new set of replication genomes
//The listlengths [2] and [3] are reset before getting here by
//createLocations or readLocations
  ga.initialize();
```

```
  while(!ga.done()) {

      //  Create children and toss into population...
      ga.step();   //nncounter++;

      //  ...evaluate aggregate population to get new scores...
      for(kk=0; kk<ga.population().size(); kk++){
        ga.population().individual(kk).evaluate(gaTrue);
      }

      //  ...Check for duplications and reevaluate the new selections
      ga.screen();
      //  This reevaluation looks impressive, but generally there is only
      //  one or two duplications in an entire population per generation
      if(duper!=0) {
        for(kk=0; kk<duper; kk++){
          ga.population().individual(dups[kk]).score(100000.);
        }
      }
      duper = 0; //Reset duper

      //  ...and then cull the worst performers out of population.
      //  Also, cull triggers the generation() to increase by 1
      ga.cull();

      //  A handy marker for which generation we're in
      gens =  ga.generation();

      //  Each generation, update cumulative crossover operator survivors
      for(kk=0; kk<ga.population().size(); kk++){
        tallyUpdate(ga.population().individual(kk));
      }

      //   This writes the entire population and score every OUTCHECK gens
        if(ga.generation() % OUTCHECK == 0) {
          fcheck << "\nGeneration: " << ga.generation(); fcheck.flush();
          for(kk=0; kk<ga.population().size(); kk++){
            fcheck << "\nGenome "<<kk<<":\n" << ga.population().individual(kk);
            fcheck << "Score\t" << ga.population().individual(kk).score() << "\n";
          }
        }

      //  Each generation, reset the trailers which track survivors
      //  By separating this function from tallyUpdate, we can check the
      //  survivors by literally examining the printed population if desired.
      for(kk=0; kk<ga.population().size(); kk++){
        resetLast(ga.population().individual(kk));
      }

      //  Reset the rates but first write the values.
      if(ga.generation() % BINSRESET == 0) {
        //Write the scores
        fcheck << ga.generation() << "\t" ;
fcheck << ga.statistics().bestIndividual().score() << "\t" ;
//Write the survivors and rates
        for(kk=0; kk<TAGS; kk++) fcheck << NN[kk] << "\t";
        //Write rates, appending to the cumulative survivor output
        for(kk=0; kk<TAGS; kk++) fcheck << prob[kk] << "\t";
//Write convergence
        fcheck<< ga.convergence() << "\t";
//Write time hack
fcheck << time(0)-starttime;
fcheck << "\n";
```

```
fcheck.flush();

        resetTally();

      }

      //Check convergence; If convergence is met, stop generation process
      if(convg == 1 &&
         ga.generation() > (nconv+1) &&
         ga.convergence() <= pconv) break;

  }

  //  Record end of GA cycles
  int endtime = time(0);

  fcheck << "\nStarted at:  " << starttime << " seconds, ";
  fcheck << "ended at:  " << endtime << " seconds\n";
  fcheck << "Total time for GA was " << endtime-starttime << " seconds, ";
  fcheck << "or  " << float (endtime-starttime)/60. << " minutes\n\n";

  fcheck << "The ga completed after " << gens << " generations.\n";
  fcheck << "The ga generated:\n" << ga.statistics().bestIndividual() << "\n";
  fcheck << "Score:\n" << ga.statistics().bestIndividual().score() << "\n";

  //Record best score
  recordGenome(ga.statistics().bestIndividual(),ga_store.population().individual(kkk), kkk);
  score_list[kkk] =  ga.statistics().bestIndividual().score();
  results[kkk].score = ga.statistics().bestIndividual().score();
  results[kkk].gens = gens;

  //Write (or append) the local xy, the global slot xy, and other stuff
  //Only necessary for randomly generated locations, as patternsV.txt is
  //available for locations which were fed in from a file
  //Can use this for future analysis if necessary.
  //Note that the locations are read only once for each repeats (the first
  //time the random locations are used), but the scores, gens, etc. are
  //repeated.

  if(locgen  && (!kkk || (kkk % repeats)==0 ) ) {
    flocs.open(name5, ios::out | ios::app );//Open or append a locations file
    flocs << results[kkk].panelcomps << "\t" ;
    if(!aim) {
      flocs  << results[kkk].paneltypes << "\t";
    }
    for (i=0; i<pcbnum; i++)
              flocs << results[kkk].pcbcomp[i] << "\t" ;
    flocs << "\n";
    //Write location (and ctypes) of components in pcb local coords
    for(i=0; i<results[kkk].panelcomps; i++){
      if( i == (results[kkk].pcbcomp[comp[i].pcbid]) || !i)
flocs << comp[i].pcbid << "\n";
      flocs  << comp[i].xloc << "\t" << comp[i].yloc;
      if(!aim) flocs << "\t" << comp[i].ct;
      flocs << "\n";
    }
    //Write slot locations or feeder distances  (if not AIM)
    if(!aim){
      for(i=0; i<results[kkk].paneltypes; i++)
       flocs << slot[i].xglob << "\t" << slot[i].yglob << "\n";
      flocs << "\n";
      flocs << "\n";
    }
    flocs.flush();
```

```
    flocs.close();
  }
  //Write GA scores after the locations
  if(locgen == 1 ) {
    flocs.open(name5, ios::out | ios::app );//Open or append a locations file
    flocs << "\n" << kkk << "\t"
      << results[kkk].score << "\n" ;
    flocs << "\n";
    flocs.flush();
    flocs.close();
  }

  //Write out an autocad file for at least the first result
  if((!kkk || (kkk % repeats)==0 ) ){
    strcpy(acadname, name1); //Separate a new name file
    //Tack on the d-file id; else, same acad name each time
    //and thus only one file for all replications
    if(innum>0) strcat(acadname, index1(dfile-1));
    strcat(acadname, ".scr" ); // Add filename extension.
    best_index = kkk;
  }

  //Write for first replication, regardless
  if(!kkk){
    acadout.open(acadname); // Open an acad file for first of every replication
    WriteAcadSeq(ga.statistics().bestIndividual());
    acadout.close(); // Close the acad file
  }
  //Write start of every replication if a discrete case
  if(innum>0 && (!kkk || (kkk % repeats)==0 )){
    acadout.open(acadname); // Open an acad file for first of every replication
    WriteAcadSeq(ga.statistics().bestIndividual());
    acadout.close(); // Close the acad file
  }
  //Check for best replication thus far:
  //Note that the acad file may be re-written as necessary
  //and a new acad file is written for each new replication,
  //so there will be replication amount of acad files for each
  //program run if for deterministic cases.
  if(kkk>0 && score_list[kkk] < score_list[best_index]) {
    best_index = kkk;
    acadout.open(acadname); // Open an acad file
    WriteAcadSeq(ga.statistics().bestIndividual());
    acadout.close(); // Close the acad file

  }
  //Reset the probabilities for next replication
  for(i=0; i<TAGS; i++) prob[i] = prob_save[i];

  } //END OF REPLICATIONS

  //Write a time stamp at END Of locs file.
  if(locgen){
    flocs.open(name5, ios::out | ios::app );//Open or append a locations file
    flocs << "#" << timepoint << "\n#";
    for(i=1; i<argc; i++) flocs << argv[i] << " ";//Print the command args
    flocs << "\n";
    flocs.flush();
    flocs.close();
  }

// Write to separate score file
  fscore.open(name0); //  Open the scores record file
  fscore << "#" << timepoint << "\n#";
```

```
  for(i=1; i<argc; i++) fscore << argv[i] << " "; //Print the command args
  fscore << "\n";
  //Write these separately, because we might break them up for analysis.
  for (i=0; i<repeats*replc; i++) {
    fscore << score_list[i] << "\n";
    if((i+1) % repeats == 0) fscore << "\n"; //Print blanks between replicatons
  }
  fscore << "\n";
  for (i=0; i<repeats*replc; i++){
//   fscore << ga_store.population().individual(i) << "\n\n";
    tempGAstore(ga_store.population().individual(i), i);
    for(j=0; j<nlists-1; j++){
      fscore << "list " << j << ": " ;
      for(k=0; k<listsizes[i][j]; k++){
        fscore << " " << templist[j][k] ;
      }
    fscore << "\n" ;
    }
    if((i+1) % repeats == 0) fscore << "\n"; //Print blanks between replicatons
  }
  fscore.close();

//  Delete the * structure pointers
  delete [] sqrect;
  delete [] comp;
  delete [] slot;
  delete [] patt;
  delete [] aa0;
  delete [] aa1;
  delete [] aa2;
  delete [] aa3;
  delete [] PCBh;
  delete [] PCBw;
  delete [] dups;
  delete [] results;

  fcheck.close();

  //I put in an exit(1) to exit gracefully and signal a 1 code upon finish.
  cout << "\nProgram end\n";
  exit(1);
  return 0;
}

// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.  This will force the instantiation
// of the template classes that we use.  For some compilers (e.g. metrowerks)
// this must come after any specializations or you'll get 'multiply-defined'
// errors when you compile.
#ifdef NO_AUTO_INST
#include <ga/GAList.C>
#include <ga/GAListGenome.C>
#if defined(__GNUG__)
template class GAList<int>;
template class GAListGenome<int>;
#else
GAList<int>;
GAListGenome<int>;
#endif
#endif
```

MYHEADER.h

```
/* MYHEADER.h
   Used in panelizer.C
   John T. Tester
   June 1999
*/

#define TOTALCOMPMAX 250         //Total allowable number of panel components
#define TOTALSLOTMAX 20         //Total allowable number of feeder slots
#define PCBMAX 8 //Total allowable number of pcb
#define PATTERNMAX 128 //Total allowable number of patterns
#define PCBENUMER  256 //Combinations for 8 rectangular PCB
#define TAGS 6 //The list size for tracking the operator survivors
#define BINS 6 //Number of survivor cumulative counters (TAGS?)
#define FLUSH 100 //Every FLUSH generations, write to trend.dat
#define OUTCHECK  6000 //Number of generations dump out genomes
#define BINSRESET  20 //Number of generations to reset counters
//(For adaptive rates calculations)
#define MINSURV 2 //The minimum number of crossover survivors per
//generation
#define POPSIZE 100               // population size
#define PERMUT 51               // Max number of panel combinations
#define MAXGENS 500             // max. number of generations
#define MAXREPLC 50 //Max number of replications
#define TRUE 1
#define FALSE 0
#define SUBSET 6 //Size of subset[] in pmaker.C

#define CLOCKS_PER_SEC 1000000   //Clocks per sec for SPARC20 (not defined in g++)
// But is found in /usr/include/time.h (Solaris)
```

Appendix B

pmaker.C

The algorithm used to build all possible panel pattern alternatives is described in section 3.1. The code, `pmaker.C`, implemented the algorithm and is constructed in 1363 lines of C++. It is presented at the end of this appendix. `pmaker.C` is a program which uses a single data file containing information about the shapes of the PCB and the number of PCB in the panel. The input format is as follows:

BOF

Panel Width, Panel Height
Number of PCB in panel
Number of unique PCB *entries*, Number of identical PCB sets *entries*
. . . Next lines are repeated for first entry of previous line
Width, Height, number (of unique PCB 1)
Width, Height, number (of unique PCB 2)
. . .
. . . Next lines are repeated for second entry of third line from top
Width, Height, number (of identical PCB set 1)
Width, Height, number (of identical PCB set 2)
. . .

EOF

An example: There are 5 PCB total in the panel. One PCB is unique (i.e., only one of its shape and/or component layout) and four PCB are identical. Though component layout may be a differentiating factor in a PCB uniqueness, the layout itself is not input here.

BOF

90. 90.
5

```
1 1
30 30 1
60 30 4
```

EOF

The results are given in the `outpatt#.dat` file, which can be renamed as `patterns#.txt`, where the `#` is the number of PCB in the panel. The latter file is then used in `panelizer` for global analysis panel design/component placement problems.

The output for the above example is listed below (A script file is also are written for drawing the patterns in Autocad). The first line of the output file specifies the total number of patterns, the second line the number of PCB in the panel. Then for each pattern, the following information is repeated:

- The pattern number (a sequential integer)
- X_{fk}, Y_{fk}, O_{fk} for each PCB in sequence

X_{fk}, Y_{fk} and O_{fk} are described in section 3.2 and refer to the PCB location and orientation of each PCB in the panel with respect to the lower-left panel corner.

```
16
5
0
-30 -30 0
-30 15 90
15 -30 0
0 15 90
30 15 90
1
-30 -30 0
-30 15 90
15 -30 0
15 0 0
15 30 0
2
-30 -30 0
-30 15 90
0 -15 90
30 -15 90
15 30 0
3
```

```
-30 30 0
-30 -15 90
15 -30 0
0 15 90
30 15 90
4
-30 30 0
-30 -15 90
15 -30 0
15 0 0
15 30 0
5
-30 30 0
-30 -15 90
0 -15 90
30 -15 90
15 30 0
6
30 -30 0
-15 -30 0
-30 15 90
0 15 90
30 15 90
7
30 30 0
-15 -30 0
-30 15 90
0 15 90
30 -15 90
8
30 -30 0
-15 -30 0
-15 0 0
-15 30 0
30 15 90
9
30 30 0
-15 -30 0
-15 0 0
-15 30 0
30 -15 90
10
30 -30 0
-30 -15 90
0 -15 90
-15 30 0
30 15 90
11
30 30 0
-30 -15 90
0 -15 90
-15 30 0
30 -15 90
12
-30 -30 0
15 -30 0
-15 0 0
-15 30 0
30 15 90
13
30 -30 0
-15 -30 0
-30 15 90
15 0 0
```

```
15 30 0
14
-30 30 0
-15 -30 0
-15 0 0
30 -15 90
15 30 0
15
30 30 0
-30 -15 90
15 -30 0
15 0 0
-15 30 0
```

pmaker.C

```
/************************************************************
pmaker.C
John T. Tester
June 8, 1999
Version 21

Creates possible panel designs based upon PCB dimensions.
************************************************************/

#include <string.h>
#include <math.h>
#include <limits.h>
#include "MYHEADER.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>

FILE *infile;
ofstream fdump;                 // Output function for write to file
ofstream acadout;                 // Output function for write to an acad script
char inname[40]; //Input filename
char outname[40];
char acadname[40];
char inpatternbase[8] = "inpatt"; //input filename base
char outpatternbase[8] = "outpatt"; //output filename base
char acadbase[8] = "acadout"; //acad output filename base
char pcb_char[2]; //A name to tack onto filenames
int nu_build; //Number of builds used during horiz/vert builds
int outnum; //A designator for output file
int innum; //A designator for input file
int allb; //0 = just patterns in acad file (default)
//1 = all builds created in acad file
int aim; //0 = AIM (default); 0 = RTHM or PAPM
int pattnum = 0; //A counter of patterns
int finalpattnum = 0; //Final number of patterns
int pcbnum = 0; //Number of pcb, total, per panel
int shapenum = 0; //Number of shapes available
int pcbunique = 0; //Number of unique pcb (.mdl == 1)
int pcbsetsame = 0; //Number like pcb (.mdl != 1) per panel
int pcbsetnum = 0; //Number of entries in pcb[] set
float panelw;
float panelh;
float PI =  3.14159265358979; //Pi

int pcbsame[8]; //No more than 8 sets (index 1) of pcb
//the same

struct builder   // build structure
{
  float dim[2]; // Aggregate [0]width/[1]ht of build
//  float DIM[2]; // Aggregate width/ht of center of build
  float area; // Area of a build
  int pcbs;   //The number of pcb which compose build
  int pcbtypcnt[PCBMAX]; //Number of pcb in build for particular pcbid
//in order from 0-PCBMAX
  int pcbid[PCBMAX]; //The pcbid which compose build
  float pcbDIM[2][PCBMAX]; //The .XX/.YY for the pcbid which is addressed
  float pcbOO[PCBMAX]; //The .OO for the pcbid which is addressed
```

```
  int subset[SUBSET]; //A check on the sub-builds which make a build
};

struct pcbinput // PCB data structure
{
  float dim[2]; // PCB [0]width/[1]ht
  int   sqrect; // Toggle on PCB square (2) or retangular (1)
  int   mdl; // Number of (this) identical pcb model in patt
};


pcbinput * pcb = new pcbinput[TOTALCOMPMAX]; //component structure
builder * build = new builder[PATTERNMAX*10]; //build structure
builder * newbuild = new builder[PATTERNMAX*5]; //temporary build structure
builder * pattern = new builder[PATTERNMAX*5]; //pattern structure (the goal)



// Set an int to a char
char setChar(int a){

  //Set the values for output file name
  switch(a){
   case 0:  pcb_char = "0";  break;
   case 1:  pcb_char = "1";  break;
   case 2:  pcb_char = "2";  break;
   case 3:  pcb_char = "3";  break;
   case 4:  pcb_char = "4";  break;
   case 5:  pcb_char = "5";  break;
   case 6:  pcb_char = "6";  break;
   case 7:  pcb_char = "7";  break;
   case 8:  pcb_char = "8";  break;
   case 9:  pcb_char = "9";  break;
default:
  cout << "\n\nIncorrect case/char number = " << a <<"\n";
  exit(1);
  }

}



//JTT  This subroutine reads pcb component input data
void readFile( void) {
  int i,j;
  if ((infile = fopen(inname,"r"))==NULL)
    {
      cout << "\nCannot open input file " << inname << "\n";
      exit(1);
    }

  fscanf(infile, "%f %f", &panelw, &panelh); //panel width and height

  //shapenum is number of DIFFERENT shapes or entries (two identical
  //shapes are counted as one entry in inpatt#.dat file).
  fscanf(infile, "%d",&shapenum); //Number of shapes (pcb)


  //Read in number of pcb sets (out of inpat#.dat entries) which are the same.
  //So, if there are 2 unique pcb and 2 other pcb (both of) which are
  //identical, then pcbunique = 2 and pcbsetsame = 1.
  //This procedure is purely for input purposes.
  fscanf(infile, "%d %d",&pcbunique, &pcbsetsame);
```

```
//  if(pcbsetsame>0)
//    for(j = 0; j < pcbsetsame; j++) fscanf(infile, "%d %d",&pcbsame[j]);


  //At this point, we could have a mix of identical and non-identical
  //PCB (ref .mdl).  Assumption is that the input file listed PCB
  //for which .mdl ==1 at beginning and then the rest were listed at
  //end.  Only one entry for each pcbsetsame

  j = 0; //Needed for pcbsetsame loop
//If no pcbunique
  if(pcbunique>0) //If at least one unique PCB
  for (j = 0; j < pcbunique; j++){
    fscanf(infile, "%f", &(pcb[j].dim[0]) );//Read PCB width
    fscanf(infile, "%f", &(pcb[j].dim[1] ) );//Read PCB height
    fscanf(infile, "%d", &(pcb[j].mdl ) );//Read PCB model population
    if(pcb[j].mdl != 1) {
      cout << "\nUnique PCB # " << j << "has more than one in panel\n" ;
      fclose(infile);
      exit(1);
    }
    pcbnum++;
  }

  //Note that i keys off last j value to keep rolling the pcb[i] index
  if(pcbsetsame>0) //At least one set of same PCB
  for (i = j; i < j+pcbsetsame; i++){
    fscanf(infile, "%f", &(pcb[i].dim[0]) );//Read PCB width
    fscanf(infile, "%f", &(pcb[i].dim[1] ) );//Read PCB height
    fscanf(infile, "%d", &(pcb[i].mdl ) );//Read PCB model population
    pcbnum += pcb[i].mdl;
    if(pcb[i].mdl < 2) {
      cout << "\nSame-shape PCB # << i << has one or less pcb per panel\n";
      fclose(infile);
      exit(1);
    }
  }

  if(pcbnum > PCBMAX) {
      cout << "\npcbnum > PCBMAX\n";
      fclose(infile);
      exit(1);
      }

  //Determine whether square or rectangular (no fixed case here)
  for (j=0; j<shapenum; j++) {
    if(pcb[j].dim[0] == pcb[j].dim[1]) pcb[j].sqrect = 2;
    else pcb[j].sqrect = 1;
  }

  fclose(infile);

}


//Initialize builds (wastes space, but not a big deal)
void
initBuilds( void )
{
  int i,j;
  for(i=0; i<PATTERNMAX*5; i++){
    build[i].dim[0]    = 0;
    build[i].dim[1]     = 0;
    build[i].pcbs   = 0;
```

```
    build[i].area   = 0.;
//    build[i].DIM[0]     = 0.;
//    build[i].DIM[1]     = 0.;
    for(j=0; j< PCBMAX; j++){
      build[i].pcbid[j] = 999; //Use dummy number instead of 0
      build[i].pcbtypcnt[j] = 0;
      build[i].pcbDIM[0][j]  = 0.;
      build[i].pcbDIM[1][j]  = 0.;
      build[i].pcbOO[j]    = 0.;
    }
    for (j=0; j<SUBSET; j++) build[i].subset[j] = 999;
  }
}

//This subroutine checks for duplicates of patterns if an AIM machine
//is under consideration (aim = 1).
//Here, no value is returned.   Instead, a rotation is marked with a
//-1 value in pattern[i].subset[SUBSET-1]
void checkAhead( int start )
{
  int i, j, k, ii, jj;
  int samepcbs = 0;
  int tally = 0; //tally of pcb duplicates per build
  int idlocator[PCBMAX]; //Array to match newbuild to build
  int duplicate = 0; //duplicate marker
  int rotnum = 0; //Number of rotations for panel

  samepcbs = 0;
  tally =0;

  builder * protated = new builder[4]; //A temporary object to hold rotations

  //set prorated to values; all same for now
  for(i=0; i<4; i++){
    protated[i].dim[0] = pattern[start].dim[0];
    protated[i].dim[1] = pattern[start].dim[1];
    protated[i].pcbs   = pattern[start].pcbs;
    for(j=0; j< PCBMAX; j++){
      protated[i].pcbtypcnt[j] = pattern[start].pcbtypcnt[j];
    }
    for(j=0; j< PCBMAX; j++){
      protated[i].pcbid[j]      = pattern[start].pcbid[j];
      protated[i].pcbDIM[0][j]  = pattern[start].pcbDIM[0][j];
      protated[i].pcbDIM[1][j]  = pattern[start].pcbDIM[1][j];
      protated[i].pcbOO[j]      = pattern[start].pcbOO[j];
    }
      for(j=0; j< SUBSET; j++){
        protated[i].subset[j]   = pattern[start].subset[j];
      }
  }


  //Square panels have 4 rotations, but rectangular have only 2
  //If rectangular, only check for 180 rotation
  if(panelw!=panelh){
    rotnum = 2-1;
    for(j=0; j< PCBMAX; j++){
      protated[0].pcbDIM[0][j] = - pattern[start].pcbDIM[0][j];
      protated[0].pcbDIM[1][j] = - pattern[start].pcbDIM[1][j];
    }
  }
  //If square, rotate around every 90 degrees.  So, check from previous
  //setting of protate[], through [0]90, [1]180 and [2]270
  else{
```

```
  rotnum = 4-1;
  for(j=0; j< PCBMAX; j++){
    protated[0].pcbDIM[0][j] = -pattern[start].pcbDIM[1][j];
    protated[0].pcbDIM[1][j] =  pattern[start].pcbDIM[0][j];
  }
  for(i=1; i<rotnum; i++){
    for(j=0; j< PCBMAX; j++){
      protated[i].pcbDIM[0][j] = -protated[i-1].pcbDIM[1][j];
      protated[i].pcbDIM[1][j] =  protated[i-1].pcbDIM[0][j];
    }
  }
}


  //This procedure is similar to checkNewBuild, except we only check
  //the pattern[] database and only from the pattern under review to
  //pattnum
  //Don't start with pattern under check (hence start+1)
  for(i = start+1; i<pattnum; i++){

    //For each pattern examined ahead of pattern[start], check
    //againstall rotations of pattern[start] (hence protated[])
    for(k = 0; k<rotnum; k++){

    //First of all, if pattern[i].subset[SUBSET-1] == -1, then
    //the pattern has already been tagged as a rotation, so skip the
    //rest of procedure.
      if(pattern[i].subset[SUBSET-1] >= 0) {

        //Are there the same type and number of pcbs?
        for(ii = 0; ii<protated[k].pcbs; ii++){
          if(protated[k].pcbtypcnt[ii]== pattern[i].pcbtypcnt[ii]) {
            samepcbs +=  pattern[i].pcbtypcnt[ii];
          }
        }
          //If pcb population same, then check corresponding pcb locations
          //For each pattern pcbid, look for a matching pattern pcbid
          // and (if equal), check x/y location.  If that matches too, then
          // increase tally.
          //ALL this must be true in order for a pattern to be a duplicate.

          if(samepcbs==protated[k].pcbs) { //Same # pcbs and types

            for(ii = 0; ii<protated[k].pcbs; ii++){ //each pcbid at a time
              for(jj = 0; jj<pattern[i].pcbs; jj++){   //vs. another

                 if(protated[k].pcbid[ii]== pattern[i].pcbid[jj] &&
                   protated[k].pcbDIM[0][ ii ]
                   ==  pattern[i].pcbDIM[0][ jj ] &&
                    protated[k].pcbDIM[1][ ii ]
                   ==  pattern[i].pcbDIM[1][ jj ] )
                       tally++;
              }
            }
          }
          else  //If samepcb!=pattern[].pcbs...
          {
            samepcbs = 0;
            tally =0;
          }
      if(tally==protated[k].pcbs) duplicate = 1;
      samepcbs = 0;
      tally =0;
```

```
      } //End of if(already struck)

    if(duplicate)  //If a duplicate, mark the pattern for
      pattern[i].subset[SUBSET-1] = -1; //latter exclusion from valid patterns
    } //End of protated alternatives
  duplicate = 0; //Reset duplicate after all rotations
  }

  delete [] protated;

}

//Check to see if the latest newbuild is not a duplication of
//earlier builds
//This returns a 1 for "not a duplication, so add to builds -- do the kkk++"
//          or a 0 for "yes a duplication so don't add to builds -- do not kkk++"
int checkNewBuild( int kkk )
{
  int i, j, ii, jj;
  int samepcbs = 0;
  int tally = 0; //tally of pcb duplicates per build
  int idlocator[PCBMAX]; //Array to match newbuild to build
  int duplicate = 0; //duplicate marker

  //The build is not a duplicate if the pcbid's and their centers (.dim[])
  // are not equal
  //Outside loop is for comparison against each build already in existence
  for(i = 0; i<nu_build; i++){
    //First see if all the pcbs are the same in newbuild vs. a build
    //Check for all pcbs in newbuild same as a given build
    //This procedure below accounts for identical pcbs in a panel
    //ii from 0 to .pcbs is imperative, because 999 for ii > .pcbs
    for(ii = 0; ii<newbuild[kkk].pcbs; ii++){
      if(newbuild[kkk].pcbtypcnt[ii]== build[i].pcbtypcnt[ii]) {
        samepcbs +=  build[i].pcbtypcnt[ii];
      }
    }
      //If pcb population the same, then check corresponding pcb locations
      //Basically, for each newbuild pcbid, look for a matching build pcbid
      // and (if equal), check x/y location.  If that matches too, then
      // increase tally.
      //ALL this must be true in order for a pattern to be a duplicate.
      if(samepcbs==newbuild[kkk].pcbs) { //Same # pcbs and types

        for(ii = 0; ii<newbuild[kkk].pcbs; ii++){ //each pcbid at a time

          for(jj = 0; jj<build[i].pcbs; jj++){       //when matched against build

            if(newbuild[kkk].pcbid[ii]== build[i].pcbid[jj] &&
              newbuild[kkk].pcbDIM[0][ ii ]
              ==  build[i].pcbDIM[0][ jj ] &&
               newbuild[kkk].pcbDIM[1][ ii ]
              ==  build[i].pcbDIM[1][ jj ] )

                  tally++;
          }
        }
      }
      else
      {
        samepcbs = 0;
        tally =0;
      }
  if(tally==newbuild[kkk].pcbs) duplicate = 1;
```

```
//  if(duplicate) break;
  samepcbs = 0;
  tally =0;

  }

  samepcbs = 0;
  tally =0;

  for(i = 0; i<kkk; i++){
    //First see if all the pcbs are the same in newbuild vs. another newbuild
    //Check for all pcbs in newbuild same as a given newbuild
    //This procedure below accounts for identical pcbs in a panel
    //ii from 0 to .pcbs is imperative, because 999 for ii > .pcbs
    for(ii = 0; ii<newbuild[kkk].pcbs; ii++){
      if(newbuild[kkk].pcbtypcnt[ii]== newbuild[i].pcbtypcnt[ii]) {
        samepcbs +=  newbuild[i].pcbtypcnt[ii];
      }
    }
      //If pcb population the same, then check corresponding pcb locations
      //Basically, for each newbuild pcbid, look for a matching newbuild pcbid
      // and (if equal), check x/y location.  If that matches too, then
      // increase tally.
      //ALL this must be true in order for a pattern to be a duplicate.

      if(samepcbs==newbuild[kkk].pcbs) { //Same # pcbs and types

        for(ii = 0; ii<newbuild[kkk].pcbs; ii++){ //each pcbid at a time

          for(jj = 0; jj<newbuild[i].pcbs; jj++){     //when matched vs. another

            if(newbuild[kkk].pcbid[ii]== newbuild[i].pcbid[jj] &&

              newbuild[kkk].pcbDIM[0][ ii ]
              ==  newbuild[i].pcbDIM[0][ jj ] &&
               newbuild[kkk].pcbDIM[1][ ii ]
              ==  newbuild[i].pcbDIM[1][ jj ] )

                  tally++;
          }
        }
      }
      else  //If samepcb!=newbuild[].pcbs...
      {
        samepcbs = 0;
        tally =0;
      }
  if(tally==newbuild[kkk].pcbs) duplicate = 1;
//  if(duplicate) break;
  samepcbs = 0;
  tally =0;

  }
  if(duplicate) return 0; //if a duplicate, return a 0.
  else return 1; //else, o.k. and return a 1.

}

//Creates a new (though temporary) HORIZONTAL build
//IMPORTANT:  kkk is incremented OUTSIDE this function.
void createNewBuildHoriz( int base, int OObase, int add, int OOadd, int kkk )
{
  int OObaseN, OOaddN, i, j, k, ii, q;
  float bw, bh, aw, ah;
```

```
  int alpha;
  if(OObase) OObaseN = 0; else OObaseN = 1;
  if(OOadd) OOaddN = 0; else OOaddN = 1;

  //Next two definitions do not require an arrary; only one of each
  builder * basebuild = new builder[2]; //A temporary object for base build
  builder * addbuild = new builder[2]; //A temporary object for add build

  float pcbtemp[2][PCBMAX]; //A temporary array for pcb dim

  for(j=0; j< PCBMAX; j++){
    pcbtemp[0][j] = 0.;
    pcbtemp[1][j] = 0.;
  }

  //set basebuild to values; keeps from corrupting the build[] database
  basebuild[0].dim[0]    = build[base].dim[0];
  basebuild[0].dim[1]     = build[base].dim[1];
  basebuild[0].pcbs   = build[base].pcbs;
  basebuild[0].area   = build[base].area;
  for(j=0; j< PCBMAX; j++){
    basebuild[0].pcbtypcnt[j] = build[base].pcbtypcnt[j];
  }
//JTT  for(j=0; j< basebuild[0].pcbs; j++){
  for(j=0; j< PCBMAX; j++){
    basebuild[0].pcbid[j] = build[base].pcbid[j];
    basebuild[0].pcbDIM[0][j]  = build[base].pcbDIM[0][j];
    basebuild[0].pcbDIM[1][j]  = build[base].pcbDIM[1][j];
    basebuild[0].pcbOO[j]    = build[base].pcbOO[j];
  }

  //set addbuild to values; keeps from corrupting the build[] database
  addbuild[0].dim[0]    = build[add].dim[0];
  addbuild[0].dim[1]     = build[add].dim[1];
  addbuild[0].pcbs   = build[add].pcbs;
  for(j=0; j< PCBMAX; j++){
    addbuild[0].pcbtypcnt[j] = build[add].pcbtypcnt[j];
  }
//  for(j=0; j< addbuild[0].pcbs; j++){
  for(j=0; j< PCBMAX; j++){
    addbuild[0].pcbid[j] = build[add].pcbid[j];
    addbuild[0].pcbDIM[0][j]  = build[add].pcbDIM[0][j];
    addbuild[0].pcbDIM[1][j]  = build[add].pcbDIM[1][j];
    addbuild[0].pcbOO[j]    = build[add].pcbOO[j];
  }

  //OObase and OOadd are also used to key the alpha settings, below
  alpha = 0;
  //Do this only if basebuild is rotated
  if(OObase > 0) {
    alpha = 1; //OObase == 1, so build rotation must be 90
    bw = basebuild[0].dim[0];
    bh = basebuild[0].dim[1];
    basebuild[0].dim[0] = bh; //Switch width and height
    basebuild[0].dim[1] = bw; //Switch width and height
  }

  //The next is done, regardless of rotation of builds, to initialize pcbtemp[]

  q = 0; //Used to keep tabs on pcbtemp
  for(i=0; i<basebuild[0].pcbs; i++){
    if(alpha<1){
      pcbtemp[0][i] = basebuild[0].pcbDIM[0][i];
      pcbtemp[1][i] = basebuild[0].pcbDIM[1][i];
```

```
  }
  else{
    pcbtemp[0][i] = -basebuild[0].pcbDIM[1][i];
    pcbtemp[1][i] =  basebuild[0].pcbDIM[0][i];
  }
  //Next is possible because we only build on either 0 or 90 degrees
  //and pcb are only shapes (if pcbOO = 0, then add 90, but if
  //pcbOO=90 already, go back to 0 when accounting for a new build + 90)
  if(alpha+basebuild[0].pcbOO[i]>1) basebuild[0].pcbOO[i] = 0;
  else basebuild[0].pcbOO[i] = basebuild[0].pcbOO[i]+alpha;
}
q=i; //set pcbtemp counter to last i

//Now, concentrate on addbuilds; account for rotation of addbuild
alpha = 0;
//Do this only if addbuild is rotated
if(OOadd > 0) {
  alpha = 1; //OObase == 1, so must be 90
  aw = addbuild[0].dim[0];
  ah = addbuild[0].dim[1];
  addbuild[0].dim[0] = ah; //Switch width and height
  addbuild[0].dim[1] = aw; //Switch width and height
}
//The below is slightly different than above, because we are adding onto
//the pcbtemp array; look for the "q+".
for(i=0; i<addbuild[0].pcbs; i++){
  if(alpha < 1){
    pcbtemp[0][q+i] = addbuild[0].pcbDIM[0][i];
    pcbtemp[1][q+i] = addbuild[0].pcbDIM[1][i];
  }
  else{
    pcbtemp[0][q+i] = -addbuild[0].pcbDIM[1][i];
    pcbtemp[1][q+i] =  addbuild[0].pcbDIM[0][i];
  }

  //Next is possible because we only build on either 0 or 90 degrees
  //and pcb are only shapes (if pcbOO = 0, then add 90, but if
  //pcbOO=90 already, go back to 0 when accounting for a new build + 90)
  if(alpha+addbuild[0].pcbOO[i]>1) addbuild[0].pcbOO[i] = 0;
  else addbuild[0].pcbOO[i] = addbuild[0].pcbOO[i]+alpha;
}

//Now, we have the base/addbuild values (except for the dims) and the
//dims values in the pcbtemp[][] array.  We could take the time to
//set the base/addbuild dims to the pcbtemp array, but don't bother here.

//First, find the new total of pcbs in build.  Recall that we checked
//this already to make sure the max number of pcbs/tyes were not exceeded
//Also, set new width and height (height is same as old, since both equal)
newbuild[kkk].pcbs = basebuild[0].pcbs + addbuild[0].pcbs;
newbuild[kkk].dim[0] = basebuild[0].dim[0] + addbuild[0].dim[0];
newbuild[kkk].dim[1] = basebuild[0].dim[1];
newbuild[kkk].subset[0] = base;
newbuild[kkk].subset[1] = add;
newbuild[kkk].subset[2] = OObase;
newbuild[kkk].subset[3] = OOadd;
newbuild[kkk].subset[4] = 0; //0 = horizontal build

//When a horizontal build is made, only the width (pcbDIM[0]) value is changed;
//it is merely an addition (subtraction) of 1/2 the width of the
//addbuild (basebuild)
// to the pcbDIM[0] values (in this case, the pcbtemp[0][] values)
// of the basebuild (addbuild).
```

```
  for(i=0; i<basebuild[0].pcbs; i++){
    newbuild[kkk].pcbid[i] = basebuild[0].pcbid[i];
    //Doing basebuild pcbs first; recall you subtract 1/2 of addbuild's width
    newbuild[kkk].pcbDIM[0][i] = pcbtemp[0][i] - addbuild[0].dim[0]/2.;
    newbuild[kkk].pcbDIM[1][i] = pcbtemp[1][i];
    newbuild[kkk].pcbOO[i] = basebuild[0].pcbOO[i];
  }

  q = i; //Use this again

  //Now repeat for addbuild pcbs, adding on to newbuild.
  //Look for the "q+" and swap base/add build around
  for(i=0; i<addbuild[0].pcbs; i++){
    newbuild[kkk].pcbid[q+i] = addbuild[0].pcbid[i];
    //Doing basebuild pcbs first; recall you add 1/2 of basebuild's width
    newbuild[kkk].pcbDIM[0][q+i] = pcbtemp[0][q+i] + basebuild[0].dim[0]/2.;
    newbuild[kkk].pcbDIM[1][q+i] = pcbtemp[1][q+i];
    newbuild[kkk].pcbOO[q+i] = addbuild[0].pcbOO[i];
  }

  //Set the pcb type counts
  for(j=0; j< PCBMAX; j++){
    newbuild[kkk].pcbtypcnt[j] = basebuild[0].pcbtypcnt[j] + addbuild[0].pcbtypcnt[j];
  }


  //Lastly, set the areas for these builds
    newbuild[kkk].area = newbuild[kkk].dim[0]*newbuild[kkk].dim[1];

  delete [] basebuild;
  delete [] addbuild;
}


//Creates a new (though temporary) VERTICAL build
//IMPORTANT:  kkk is incremented OUTSIDE this function.
void createNewBuildVert( int base, int OObase, int add, int OOadd, int kkk )
{
  int OObaseN, OOaddN, i, j, k, ii, q;
  float bw, bh, aw, ah;
  int alpha;
  if(OObase) OObaseN = 0; else OObaseN = 1;
  if(OOadd) OOaddN = 0; else OOaddN = 1;

  //Next two definitions do not require an array; only one of each
  builder * basebuild = new builder[2]; //A temporary object for base build
  builder * addbuild = new builder[2]; //A temporary object for add build

  float pcbtemp[2][PCBMAX]; //A temporary array for pcb dim

  for(j=0; j< PCBMAX; j++){
    pcbtemp[0][j] = 0.;
    pcbtemp[1][j] = 0.;
  }

  //set basebuild to values; keeps from corrupting the build[] database
  basebuild[0].dim[0]    = build[base].dim[0];
  basebuild[0].dim[1]     = build[base].dim[1];
  basebuild[0].pcbs   = build[base].pcbs;
  for(j=0; j< PCBMAX; j++){
    basebuild[0].pcbtypcnt[j] = build[base].pcbtypcnt[j];
  }
  for(j=0; j< PCBMAX; j++){
    basebuild[0].pcbid[j] = build[base].pcbid[j];
```

```
  basebuild[0].pcbDIM[0][j]  = build[base].pcbDIM[0][j];
  basebuild[0].pcbDIM[1][j]  = build[base].pcbDIM[1][j];
  basebuild[0].pcbOO[j]    = build[base].pcbOO[j];
}

//set addbuild to values; keeps from corrupting the build[] database
addbuild[0].dim[0]    = build[add].dim[0];
addbuild[0].dim[1]     = build[add].dim[1];
addbuild[0].pcbs   = build[add].pcbs;
for(j=0; j< PCBMAX; j++){
  addbuild[0].pcbtypcnt[j] = build[add].pcbtypcnt[j];
}
for(j=0; j< PCBMAX; j++){
  addbuild[0].pcbid[j] = build[add].pcbid[j];
  addbuild[0].pcbDIM[0][j]  = build[add].pcbDIM[0][j];
  addbuild[0].pcbDIM[1][j]  = build[add].pcbDIM[1][j];
  addbuild[0].pcbOO[j]    = build[add].pcbOO[j];
}

//OObase and OOadd are also used to key the alpha settings, below
alpha = 0;
//Do this only if basebuild is rotated
if(OObase > 0) {
  alpha = 1; //OObase == 1, so build rotation must be 90
  bw = basebuild[0].dim[0];
  bh = basebuild[0].dim[1];
  basebuild[0].dim[0] = bh; //Switch width and height
  basebuild[0].dim[1] = bw; //Switch width and height
}

//The next is done, regardless of rotation of builds, to initialize pcbtemp[]

q = 0; //Used to keep tabs on pcbtemp
for(i=0; i<basebuild[0].pcbs; i++){
  if(alpha < 1){
    pcbtemp[0][i] = basebuild[0].pcbDIM[0][i];
    pcbtemp[1][i] = basebuild[0].pcbDIM[1][i];
  }
  else{
    pcbtemp[0][i] = -basebuild[0].pcbDIM[1][i];
    pcbtemp[1][i] =  basebuild[0].pcbDIM[0][i];
  }

  //Next is possible because we only build on either 0 or 90 degrees
  //and pcb are only shapes (if pcbOO = 0, then add 90, but if
  //pcbOO=90 already, go back to 0 when accounting for a new build + 90)
  if(alpha+basebuild[0].pcbOO[i]>1) basebuild[0].pcbOO[i] = 0;
  else basebuild[0].pcbOO[i] = basebuild[0].pcbOO[i]+alpha;
}
q=i; //set pcbtemp counter to last i

//Now, concentrate on addbuilds; account for rotation of addbuild
alpha = 0;
//Do this only if addbuild is rotated
if(OOadd > 0) {
  alpha = 1; //OObase == 1, so must be 90
  aw = addbuild[0].dim[0];
  ah = addbuild[0].dim[1];
  addbuild[0].dim[0] = ah; //Switch width and height
  addbuild[0].dim[1] = aw; //Switch width and height
}
//The below is slightly different than above, because we are adding onto
//the pcbtemp array; look for the "q+".
for(i=0; i<addbuild[0].pcbs; i++){
```

```
  if(alpha < 1){
    pcbtemp[0][q+i] = addbuild[0].pcbDIM[0][i];
    pcbtemp[1][q+i] = addbuild[0].pcbDIM[1][i];
  }
  else{
    pcbtemp[0][q+i] = -addbuild[0].pcbDIM[1][i];
    pcbtemp[1][q+i] =  addbuild[0].pcbDIM[0][i];
  }

  //Next is possible because we only build on either 0 or 90 degrees
  //and pcb are only shapes (if pcbOO = 0, then add 90, but if
  //pcbOO=90 already, go back to 0 when accounting for a new build + 90)
  if(alpha+addbuild[0].pcbOO[i]>1) addbuild[0].pcbOO[i] = 0;
  else addbuild[0].pcbOO[i] = addbuild[0].pcbOO[i]+alpha;
}

//Now, we have the base/addbuild values (except for the dims) and the
//dims values in the pcbtemp[][] array.  We could take the time to
//set the base/addbuild dims to the pcbtemp array, but don't bother here.

//First, find the new total of pcbs in build.  Recall that we checked
//this already to make sure the max number of pcbs/types were not exceeded
newbuild[kkk].pcbs = basebuild[0].pcbs + addbuild[0].pcbs;

//Also, set new width and height
//NOTE: This is first change relative to createNewHoriz
newbuild[kkk].dim[0] = basebuild[0].dim[0];
newbuild[kkk].dim[1] = basebuild[0].dim[1] + addbuild[0].dim[1];
newbuild[kkk].subset[0] = base;
newbuild[kkk].subset[1] = add;
newbuild[kkk].subset[2] = OObase;
newbuild[kkk].subset[3] = OOadd;
newbuild[kkk].subset[4] = 1; //1 = vertical build

//When a VERTICAL build is made, only the height (pcbDIM[1]) value is changed;
//it is merely an addition (subtraction) of 1/2 the width of the
//addbuild (basebuild)
// to the pcbDIM[1] values (in this case, the pcbtemp[1][] values)
// of the basebuild (addbuild).

for(i=0; i<basebuild[0].pcbs; i++){
  newbuild[kkk].pcbid[i] = basebuild[0].pcbid[i];
  //NOTE:  The next 2 commands (exclude comments)
    //are changes from createNewHoriz
  //Doing basebuild pcbs first; recall you subtract 1/2 of addbuild's HEIGHT
  newbuild[kkk].pcbDIM[0][i] = pcbtemp[0][i];
  newbuild[kkk].pcbDIM[1][i] = pcbtemp[1][i] - addbuild[0].dim[1]/2.;
  newbuild[kkk].pcbOO[i] = basebuild[0].pcbOO[i];
}

q = i; //Use this again

//Now repeat for addbuild pcbs, adding on to newbuild.
//Look for the "q+" and swap base/add build around
for(i=0; i<addbuild[0].pcbs; i++){
  newbuild[kkk].pcbid[q+i] = addbuild[0].pcbid[i];
  //NOTE:  The next 2 commands (exclude comments)
    //are changes from createNewHoriz
  //Doing basebuild pcbs first; recall you add 1/2 of basebuild's width
  newbuild[kkk].pcbDIM[0][q+i] = pcbtemp[0][q+i];
  newbuild[kkk].pcbDIM[1][q+i] = pcbtemp[1][q+i] + basebuild[0].dim[1]/2.;
  newbuild[kkk].pcbOO[q+i] = addbuild[0].pcbOO[i];
}
```

```
  //Set the pcb type counts
  for(j=0; j< PCBMAX; j++){
    newbuild[kkk].pcbtypcnt[j] = basebuild[0].pcbtypcnt[j] + addbuild[0].pcbtypcnt[j];
  }

  //Lastly, set the areas for these builds
    newbuild[kkk].area = newbuild[kkk].dim[0]*newbuild[kkk].dim[1];

  delete [] basebuild;
  delete [] addbuild;
}

//Add all the new builds to main build database
void addNewBuild( int kkk )
{
  int i,j;

  for(i=0; i<kkk; i++){
    build[nu_build+i].dim[0] = newbuild[i].dim[0];
    build[nu_build+i].dim[1] = newbuild[i].dim[1];
    build[nu_build+i].pcbs   = newbuild[i].pcbs;
    build[nu_build+i].area   = newbuild[i].area;
    for(j=0; j< PCBMAX; j++){
      build[nu_build+i].pcbtypcnt[j] = newbuild[i].pcbtypcnt[j];
    }
//    for(j=0; j< newbuild[i].pcbs; j++){
    for(j=0; j< PCBMAX; j++){
      build[nu_build+i].pcbid[j] = newbuild[i].pcbid[j];
      build[nu_build+i].pcbDIM[0][j]  = newbuild[i].pcbDIM[0][j];
      build[nu_build+i].pcbDIM[1][j]  = newbuild[i].pcbDIM[1][j];
      build[nu_build+i].pcbOO[j]    = newbuild[i].pcbOO[j];
    }
    for(j=0; j< SUBSET; j++){
      build[nu_build+i].subset[j]   = newbuild[i].subset[j];
    }
  }

}

//Create the patterns database, almost a duplicate of addNewBuild(), above,
//except that we check for valid patterns first.
//This database may be expanded or contracted for the AIM vs. other cases
void createPatterns( void )
{
  int i,j;
  pattnum = 0;

  //Check to see which builds are just subsets of patterns
  //This is accomplished by calculating the areas , since all other
  //checks have been accomplished earlier

  for(i=0; i<nu_build; i++){
    if(build[i].area == panelw*panelh) {
      pattern[pattnum].dim[0] = build[i].dim[0];
      pattern[pattnum].dim[1] = build[i].dim[1];
      pattern[pattnum].pcbs   = build[i].pcbs;
      pattern[pattnum].area   = build[i].area;
      for(j=0; j< PCBMAX; j++){
        pattern[pattnum].pcbtypcnt[j] = build[i].pcbtypcnt[j];
      }
      for(j=0; j< PCBMAX; j++){
        pattern[pattnum].pcbid[j] = build[i].pcbid[j];
        pattern[pattnum].pcbDIM[0][j]  = build[i].pcbDIM[0][j];
        pattern[pattnum].pcbDIM[1][j]  = build[i].pcbDIM[1][j];
```

```
        pattern[pattnum].pcbOO[j]    = build[i].pcbOO[j];
      }
      for(j=0; j< SUBSET; j++){
        pattern[pattnum].subset[j]   = build[i].subset[j];
      }
      pattnum++;
    }
  }
}

//This writes all builds to file; hence, no check for rotations exclusions
//is accomplished in this subroutine.
void acadWriteBuilds( void )
{
  int i, j, k;
  int count = 0; //Count number of patterns written
  int columns = 1; //Number of columns
  float offsetrow = 0.; //Offset for the rows
  float offsetcol = 0.; //Offset for the columns
  int xr = 0; //x pcb dim index: 0->0 deg, 1->90 deg
  int yr = 1; //y pcb dim index: 0->0 deg, 1->90 deg

  acadout.open(acadname);
  columns = 8;

  //Make sure you have enough screen to start drawing (Acad quirk)
  acadout <<  "ZOOM\n.01x\n";

  //NEED LOOP FOR ALL BUILDS
  //WITHIN LOOP, CONTINGENT ON A IF STATEMENT ON BUILD[].AREA
  //INSIDE LOOP, PUT IN AN OFFSET VALUE BETWEEN PATTERNS.
  for( i=0; i<nu_build; i++){

    //JTT Draw rectangles for the PCB in this solution
    for( k = 0; k<build[i].pcbs; k++){
      //If a pcb is rotated, the rectangle dims will be exchanged
      if(build[i].pcbOO[k]>0) {
        xr = 1;
        yr = 0;
      }
      else {
        xr = 0;
        yr = 1;
      }

      acadout <<  "RECTANG\n";
      acadout  << -pcb[ build[i].pcbid[k] ].dim[xr]/2.0 +offsetcol<< ","
       << -pcb[ build[i].pcbid[k] ].dim[yr]/2.0 +offsetrow<< "\n";
      acadout  << pcb[ build[i].pcbid[k] ].dim[xr]/2.0 +offsetcol<< ","
          << pcb[ build[i].pcbid[k] ].dim[yr]/2.0 +offsetrow<< "\n";

      //JTT Move PCB to correct XX and YY position
      acadout << "MOVE\nLAST\n\n" << "0.0,0.0,0.0\n";
      acadout << build[i].pcbDIM[0][ k ] << ",";
      acadout << build[i].pcbDIM[1][ k ]<< "\n";

    //Write the PCBID
      acadout <<  "TEXT\n";
      acadout << build[i].pcbDIM[0][k] + offsetcol << "," << build[i].pcbDIM[1][k] + offsetrow << "\n";
      acadout  << "5.0\n0\n";
      acadout <<  k << "\n";

    } //End of pattern drawing
```

```
    //Print out build number (for check);
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 2 << "\n5.0\n0\n";
      acadout <<  "Build Number: " << i << "\n";

    //And pattern number (for final version);
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 9 << "\n5.0\n0\n";
      acadout <<  "Pattern Number: " << count << "\n";

    //And pcbid set
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 16 << "\n5.0\n0\n";
      acadout <<  "pcbid: {";
      for(k=0; k<build[i].pcbs-1; k++) acadout << build[i].pcbid[k]  << ", ";
      acadout << build[i].pcbid[k] ;
      acadout << "}\n";

    //And pcbOO
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 23 << "\n5.0\n0\n";
      acadout <<  "pcbOO: {";
      for(k=0; k<build[i].pcbs-1; k++)  acadout << build[i].pcbOO[k] << ", ";
      acadout << build[i].pcbOO[k];
      acadout << "}\n";

    //And subsets
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 30 << "\n5.0\n0\n";
      acadout <<  "ss: {";
      for(k=0; k<SUBSET-1; k++) {
        if(build[i].subset[k] > 998) acadout << "-, ";
        else acadout << build[i].subset[k] << ", ";
      }
      if(build[i].subset[k] > 998) acadout << "-";
      else acadout << build[i].subset[k];
      acadout << "}\n";

  if(build[i].area == panelw*panelh) count++;
    offsetcol += 1.5*panelw; //Push over for next build

    if ( i!=0 && (((i+1) % columns) == 0) ){//Reset for next row of patterns
      offsetrow += 1.5*panelh;
      offsetcol = 0.;
    }
  }
  // Print zoom commands to finish
  acadout <<  "ZOOM\nAll\n";

  acadout.close();
}

void acadWritePatterns( void )
{
  int i, j, k;
  int pcount = 0; //Count number of patterns written
  int columns = 8; //Number of columns
  float offsetrow = 0.; //Offset for the rows
  float offsetcol = 0.; //Offset for the columns
```

```
int xr = 0; //x pcb dim index: 0->0 deg, 1->90 deg
int yr = 1; //y pcb dim index: 0->0 deg, 1->90 deg

acadout.open(acadname);

//Figure out a grid for offset drawings
if(finalpattnum > 48 ) columns = 16;
else if(finalpattnum > 28 ) columns = 8;
else if(finalpattnum > 12 ) columns = 4;
else if(finalpattnum > 40) columns = 2;


//Make sure you have enough screen to start drawing (Acad quirk)
acadout <<  "ZOOM\n.01x\n";

//NEED LOOP FOR ALL BUILDS
//WITHIN LOOP, CONTINGENT ON A IF STATEMENT ON BUILD[].AREA
//INSIDE LOOP, PUT IN AN OFFSET VALUE BETWEEN PATTERNS.
for( i=0; i<pattnum; i++){

  if( pattern[i].subset[SUBSET-1] >= 0) { //Only if not marked

    //JTT Draw rectangles for the PCB in this solution

    for( k = 0; k<pattern[i].pcbs; k++){

      //If a pcb is rotated, the rectangle dims will be exchanged
      if(pattern[i].pcbOO[k]>0) {
          xr = 1;
          yr = 0;
        }
        else {
          xr = 0;
          yr = 1;
      }

      acadout <<  "RECTANG\n";
      acadout  << -pcb[ pattern[i].pcbid[k] ].dim[xr]/2.0 +offsetcol<< ","
       << -pcb[ pattern[i].pcbid[k] ].dim[yr]/2.0 +offsetrow<< "\n";
      acadout  << pcb[ pattern[i].pcbid[k] ].dim[xr]/2.0 +offsetcol<< ","
          << pcb[ pattern[i].pcbid[k] ].dim[yr]/2.0 +offsetrow<< "\n";

      //JTT Move PCB to correct XX and YY position
      acadout << "MOVE\nLAST\n\n" << "0.0,0.0,0.0\n";
      acadout << pattern[i].pcbDIM[0][ k ] << ",";
      acadout << pattern[i].pcbDIM[1][ k ]<< "\n";

    //Write the PCBID
      acadout <<  "TEXT\n";
      acadout << pattern[i].pcbDIM[0][k] + offsetcol << "," << pattern[i].pcbDIM[1][k] + offsetrow << "\n";
      acadout  << "5.0\n0\n";
      acadout <<  k << "\n";

    } //End of pattern drawing


    //And pattern number (for final version);
    acadout <<  "TEXT\n";
    acadout << offsetcol - 0.5*panelw<< ","
         << offsetrow - 0.5*panelh - 9 << "\n5.0\n0\n";
    acadout <<  "Pattern Number: " << pcount << "\n";

    //And pcbid set
    acadout <<  "TEXT\n";
```

```
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 16 << "\n5.0\n0\n";
      acadout <<  "pcbid: {";
      for(k=0; k<pattern[i].pcbs-1; k++) acadout << pattern[i].pcbid[k]  << ", ";
      acadout << pattern[i].pcbid[k] ;
      acadout << "}\n";

      //And pcbOO
      acadout <<  "TEXT\n";
      acadout << offsetcol - 0.5*panelw<< ","
             << offsetrow - 0.5*panelh - 23 << "\n5.0\n0\n";
      acadout <<  "pcbOO: {";
      for(k=0; k<pattern[i].pcbs-1; k++)  acadout << pattern[i].pcbOO[k] << ", ";
      acadout << pattern[i].pcbOO[k];
      acadout << "}\n";

      pcount++;
      offsetcol += 1.5*panelw; //Push over for next pattern
      if ( i!=0 && i % columns == 0 ){ //Reset for next row of patterns
        offsetrow += 1.5*panelh;
        offsetcol = 0.;
      }

    }

  }
  // Print zoom commands to finish
  acadout <<  "ZOOM\nAll\n";

  acadout.close();

}

int
main(int argc, char *argv[])
{
  int i, j, ii, kkk, sucess, OObase, OOadd, OObaseN, OOaddN, jlow;
  int bumplimit;
  int pcount = 0;
//  int bumpup = 0;
//  int oldbumpvalue = 0;
  outnum = 0;
  innum  = 0;
  allb   = 0;
  aim    = 0;
  finalpattnum = 0;


  for(i=1; i<argc; i++){
    if(strcmp("innum", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": number for text input file needs a value.\n";
        exit(1);
      }
      else{
        innum = atoi(argv[i]);
        continue;
      }
    }
    if(strcmp("outnum", argv[i]) == 0){
      if(++i >= argc){
        cerr << argv[0] << ": number for text output file needs a value.\n";
        exit(1);
      }
```

```
    else{
      outnum = atoi(argv[i]);
      continue;
    }
  }

  if(strcmp("allb", argv[i]) == 0){
    if(++i >= argc){
      cerr << argv[0] << ": alb (all builds) needs a value.\n";
      exit(1);
    }
    else{
      allb = atoi(argv[i]);
      continue;
    }
  }

  if(strcmp("aim", argv[i]) == 0){
    if(++i >= argc){
      cerr << argv[0] << ": aim flag needs a value.\n";
      exit(1);
    }
    else{
      aim = atoi(argv[i]);
      continue;
    }
  }

  else {
    cerr << argv[0] << ":  unrecognized argument: " << argv[i] << "\n\n";
    exit(1);
  }

}

setChar(innum);
strcat( inname, inpatternbase );
strcat( inname, pcb_char );
strcat( inname, ".dat" );

setChar(outnum);
strcat( outname, outpatternbase );
strcat( outname, pcb_char );
strcat( outname, ".dat" );

setChar(outnum);
strcat( acadname, acadbase );
strcat( acadname, pcb_char );
strcat( acadname, ".scr" );

fdump.open(outname); //Open up here to use for checking other stuff

//Read in pcb and panel data for patterns
readFile();

//Initialize possible build values
initBuilds();

//First, define all pcb (types) as valid builds.  Thus, the first
//shape numwill be the basic pcb shapes.
for(i=0; i<shapenum; i++) {
  build[i].pcbs = 1; //This is initialization
  build[i].pcbid[0] = i;
  build[i].pcbtypcnt[i] = 1; //All that's needed; rest are 0(from init)
```

```
    build[i].dim[0] = pcb[i].dim[0];
    build[i].dim[1] = pcb[i].dim[1];
    build[i].pcbDIM[0][0] = 0.; //A pcb's center by itself is at 0.
    build[i].pcbDIM[1][0] = 0.; //A pcb's center by itself is at 0.
    build[i].pcbOO[0] = 0; //initially, pcb at 0 degrees
  }

  nu_build = i; // After initialization, the number of builds
// will be the number of pcb in panel

  bumplimit = 0; //Initial setting for this limit

  //We do builds by adding horizontal or vertical distances.
  //If either exceeds panelh or panelw, invalid build
  //If above passes, then if buildh == add ht (there is no waste), so go on
  //Tally the base + add pcb populations (types) and if within valid ranges,
  //create a new build.


  kkk = 0;
  OObase = 0;
  OOadd = 0;

  //*** Probably, here the do loop for horiz/vertical builds starts
  //*** Should be contingent on continuing as long as kkk != 0 at end of loop.
  //*** Therefore, kkk=0 inside and right after do { statement

  do {

  kkk = 0;

  //Begin first horizontal build

  for(OObase = 0; OObase<2; OObase++){ //This increments OObase
  for(OOadd = 0; OOadd<2; OOadd++){ //This increments OOadd

  //OObase is the (default) width of a base build.
  //If OObase = 1, then it is rotated and that value in a dim[] can be
  //the ht value for the base, used for the width.
  //OObaseN is the orthogonal dimension of the width, which would be the ht.
  //It must be set opposite to OObase, or 1 when OObase is 0 and vice-versa.
  //Ditto for OOadd and OOaddN, but applied to the addbuild, or build[j], below.
  if(OObase) OObaseN = 0; else OObaseN = 1; // May be rotating OObase around
  if(OOadd) OOaddN = 0; else OOaddN = 1; // May be rotating OOadd around

  sucess = 1; // sucess = 0 means too many pcb for a newbuild

  for(i=0; i<nu_build; i++){

    if(i<bumplimit) jlow = bumplimit; //Prevents duplication of earlier builds
    else jlow = 0;

    for(j=jlow; j<nu_build; j++){  //This "for" prevents mirrors

      //Check:  sum of horizontal build dims <= panelw and
      //        build[base] ht == build[add] ht, then (note:  ht is the OO_N)
      //check pcb population totals before the newbuild call.
      if( (build[i].dim[OObase]+build[j].dim[OOadd]) <= panelw &&  //key is panelw
        (build[i].dim[OObaseN] == build[j].dim[OOaddN]) && //no waste
        (build[i].dim[OObaseN] <= panelh) ){ //fits ht
          for(ii = 0; ii<pcbnum; ii++){
  if( (build[i].pcbtypcnt[ii] + build[j].pcbtypcnt[ii]) > pcb[ii].mdl )
     sucess = 0; // Too many pcb for a new build
      }
```

```
        if(sucess) {
  createNewBuildHoriz(i, OObase, j, OOadd, kkk);
          //After all this work, we now need to see if this is actually
          //a duplicate of an earlier build.  If so, don't bother to ADD
          //the creation to the build set (it will be overwritten by the
          //next creation)
  if(checkNewBuild(kkk)) kkk++;// Increment the (temporary) newbuild
        }
       }
    sucess = 1;

    }
  }

} //ends OOadd loop
} //ends OObase loop

//Repeat above for Vertical builds.  Look at panelw as well as
//the dim[OObase]->dim[OObaseN] and so on.

  //Begin first VERTICAL build

  OObase = 0;
  OOadd = 0;

  for(OObase = 0; OObase<2; OObase++){ //This increments OObase
  for(OOadd = 0; OOadd<2; OOadd++){ //This increments OOadd

  //OObase is the (default) width of a base build.
  //If OObase = 1, then it is rotated and that value in a dim[] can be
  //the ht value for the base, used for the width.
  //OObaseN is the orthogonal dimension of the width, which would be the ht.
  //It must be set opposite to OObase, or 1 when OObase is 0 and vice-versa.
  //Ditto for OOadd and OOaddN, but applied to the addbuild, or build[j], below.
  if(OObase) OObaseN = 0; else OObaseN = 1; // May be rotating OObase around
  if(OOadd) OOaddN = 0; else OOaddN = 1; // May be rotating OOadd around

  sucess = 1; // sucess = 0 means too many pcb for a newbuild

  for(i=0; i<nu_build; i++){

    if(i<bumplimit) jlow = bumplimit; //Prevents duplication of earlier builds
    else jlow = 0;

    for(j=jlow; j<nu_build; j++){  //This "for" prevents mirrors

      //Check:  sum of VERTICAL build dims <= panelh and
      //        build[base] wth == build[add] wth, then (note:  wth is the OO_N)
      //check pcb population totals before the newbuild call.
      //The changes between the HORIZONTAL build and VERTICAL build section
      //is here:  switch OObase<-->OObaseN and same for OOadd.
      //Also panelw-->panelh
      if( (build[i].dim[OObaseN]+build[j].dim[OOaddN])<=panelh && //key is panelh
        (build[i].dim[OObase] == build[j].dim[OOadd]) && //no waste
        (build[i].dim[OObase]) <= panelw ){ //fits width
          for(ii = 0; ii<pcbnum; ii++){
  if( (build[i].pcbtypcnt[ii] + build[j].pcbtypcnt[ii]) > pcb[ii].mdl )
       sucess = 0; // Too many pcb for a new build
        }
        if(sucess) {
          //A different creation function for Vertical vs. Horizontal builds
  createNewBuildVert(i, OObase, j, OOadd, kkk);
          //After all this work, we now need to see if this is actually
          //a duplicate of an earlier build.  If so, don't bother to ADD
```

```
          //the creation to the build set (it will be overwritten by the
          //next creation)
  if(checkNewBuild(kkk)) kkk++;// Increment the (temporary) newbuild
        }
       }
    sucess = 1;

    }
  }

} //ends OOadd loop
} //ends OObase loop



  //Set the bumplimit to the old (current) nu_build value and reset bumpup
  bumplimit = nu_build;
  //Now add the newbuilds to the valid build sets
  if(kkk!=0) {
    addNewBuild( kkk );
    nu_build += kkk;
  }

  }while(kkk!=0); //end of "do"



  //Now create patterns database.
  createPatterns();

  int finder = 0; //For tracking dups output

  //If an AIM case, we need to reduce the patterns set to eliminate
  //rotations of the same pattern designs
  if(aim) for(i=0; i<pattnum-1; i++) checkAhead(i);

  //Now, all duplicates are marked for deletion from database.
  //Instead of rewriting the pattern[] database, just check
  //each pattern's subset[SUBSET-1] mark before output.
  //But, we need a new pattern count: finalpattnum
  if(aim) {
    for(i=0; i<pattnum; i++) {
      if( pattern[i].subset[SUBSET-1] >= 0) finalpattnum++;
    }
  }
  else finalpattnum = pattnum;


  //First, print to an outfile, only patterns
  fdump << finalpattnum << "\n" << pcbnum << "\n";
  for(i=0; i<pattnum; i++){

    if( pattern[i].subset[SUBSET-1] >= 0) { //Only if not marked
      fdump << pcount << "\n"; //pattern number
      pcount++;
      //Must account for duplicate pcb in pattern, so use shapenum again
      //Note:  You don't need to keep the order of the identical shapes
      //the same, for how do you tell the difference?  They are identical...
      //Oh, and add 1/2 of panel width/ht to output, for reference to
      //Lower-left hand corner.
      for(j=0; j<shapenum; j++){
        for(ii=0; ii<pattern[i].pcbs; ii++){
  if(pattern[i].pcbid[ii]==j){
            fdump << pattern[i].pcbDIM[0][ii]+0.5*panelw << "\t";
```

```
    fdump << pattern[i].pcbDIM[1][ii]+0.5*panelh << "\t";
            if(pattern[i].pcbOO[ii]<1) fdump << 0 ;
            else fdump << 90 ;
            fdump << "\n";
          }
        }
      }
    }

  }
  fdump.close();

  //Now, print out an autocad file to check.
  if(allb > 0) acadWriteBuilds();
  else acadWritePatterns();

  delete [] pcb;
  delete [] build;
  delete [] newbuild;
  delete [] pattern;


}
```

Appendix C

`finder.C`

This appendix briefly introduces the `finder.C` program, which estimates the subjective value of all possible panel designs for a set of pattern alternatives and PCB component layouts via the estimator functions described in section 3.5. It uses the `patterns#.txt`, the `dP?.txt` and the `<machine>pcbV#.txt` files. The "#" is the number of PCB in the panel, the "?" is a file designator (just an integer between 0 and 20) and the "`<machine>`" prefix represents the machine type under examination.

The patterns#.txt file contains the g_f information, consisting of the $X_{fk}i$, Y_{fk} and O_{fk} for each pattern. The dP?.txt file (or dA?.txt file for AIM) contains the x, y coordinates for all the components for each PCB in the panel, relative to the geometric center of each PCB. The component type for each component is supplied as a third argument for PAPM and RTHM cases. The slot locations are provided below all the components information for the PAPM and RTHM; the AIM does not need that information. The `<machine>pcbV#.txt` file gives the number of PCB on the panel as well as the width and height of each PCB. The number of heads, table speed, linear rack speed and index time is also required at the bottom of this file for the RTHM experiments.

The program is interactive and thus self-explanatory, given the above files are present in the same directory as that of the `finder.C` program.

`finder.C` is implemented in 1379 lines of C++ code, provided in this appendix. It uses the same header file as that of `panelizer` (Appendix D).

finder.C

```
/*************************************************************************
finder.C b.16
John Tester 1999
Version 16

This program enumerates all rotations
for a pattern file.  The enumerations are used to calculate the
fast approximation of the "goodness" of the rotation situation for
component placement.  It also calculates the worst situation.  Note that
there can be multiple good and bad situations with the same or similar
scores.  Only the first one of each is recorded, though you can write all
enumerations to file if desired.

Note that this program does not allow for any fixed pcb shapes.  If you want
to do that, need to alter bottom of readPCBfile()

*************************************************************************/
#include <string.h>
#include <math.h>
#include "MYHEADER.h"
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>


unsigned long getindex (void);
int getrot (int );

int pcbtype[32], pcbcount, pcbrot[32];
int sqrect[PCBMAX];  //Shape array
int switcher = 1;
int sizer = 0;
int kkk = 0; //Pattern tracker
unsigned long stringer = 0;

int totalcomps = 0;

char prompt; //A dummy used for prompt input
char pattfile[40]; //Pattern file input name (discrete cases)
char pcbfile[40]; //PCB file input name
char locations[40]; //PCB file input name
int lindex; //Index for dA/dP .txt files
char name1[4] = "Est"; //Output file name prefix
char name0[5] = "Af"; //Acad script name prefix
char outfile[40]; //Output (estimates)
char exfile[40]; //Output (estimates) for excel file
char acadfilebest[40]; //Acad script filename
char acadfileworst[40]; //Acad script filename
char bigdump; //Flag for output file dump of all information
char smalldump; //Flag for output file dump of only best/worst
char exceldump; //Flag for output file dump of an Excel-oriented file
char acadbest; //Flag for output to acad best
char acadworst; //Flag for output to acad worst
char * timepoint; //Used to catch time for files writing

int pcbnum;                     //Total number of pcb
int racknum;                    //Total rack slots
```

```
int pattnum;                    //Total number of patterns
float slotspace = 0.;           //Spacing between slots
int heads = 12;                 //Number of heads for RTHM
float indextime = 0.15;         //Index time (sec/component) for RTHM
float slottime = 0.15;          //Feeder slot time (sec/slots moved) for RTHM
float tablevel = 1.0;           //Table velocity (mm/sec) for RTHM

int pcb_rot_best[PCBMAX]; //Array which contains best pcb orientations
int pcb_rot_worst[PCBMAX]; //Array which contains worst pcb orientations
float pnlcntr[2]; //Temporary panel centriod
float pnlcntr_best[2]; //Best panel center
float pnlcntr_worst[2]; //Worst panel center
float cntrs_bestx[PCBMAX][TOTALSLOTMAX];//The centers of the sub populations
float cntrs_besty[PCBMAX][TOTALSLOTMAX];//The centers of the sub populations
float cntrs_worstx[PCBMAX][TOTALSLOTMAX];//The centers of the sub populations
float cntrs_worsty[PCBMAX][TOTALSLOTMAX];//The centers of the sub populations
float comx[PCBMAX][TOTALSLOTMAX];//Temporary
float comy[PCBMAX][TOTALSLOTMAX];//Temporary
int typecount[PCBMAX][TOTALSLOTMAX];//Number of each comp type in each pcb
int assign[TOTALSLOTMAX]; //assign[slot location] = ctype
int perm_assign[TOTALSLOTMAX]; //permenant assign[slot location] = ctype; refreshes between iterations
int ct_order[TOTALSLOTMAX]; //Component types in increasing order of population
float magnitude; //Temporary estimator magnitude value
float initbest; //Initial estimate value
float magbest; //Best estimate value
float magworst; //Worst estimate value
int pattbest; //Best estimate pattern
int pattworst; //Worst estimate pattern

int aim = 0;                    //Set aim case:  0 for no aim, 1 for aim
int papm = 1;                   //Set papm case: 0 for no papm, 1 for papm (default)
int rthm = 0;                   //Set rthm case:  0 for no rthm, 1 for rthm
float rcx[TOTALSLOTMAX];        //Measure for centers of component population(s)
float rcy[TOTALSLOTMAX];        //Measure for centers of component population(s)
float type[TOTALSLOTMAX];       //Component types recorded (as necessary)
float Rc[TOTALSLOTMAX];         //Magnitude of rx/y center(s)

int pcbcomps[PCBMAX]; //Number of components for a PCB
float xsum[PCBMAX][TOTALSLOTMAX];//CUmulative sum of x-coords for a PCB comp pop group
float ysum[PCBMAX][TOTALSLOTMAX];//CUmulative sum of y-coords for a PCB comp pop group



int * aa0 = new int[PCBMAX];    // The rotation part 0; depends on the sublist[1]
int * aa1 = new int[PCBMAX];
int * aa2 = new int[PCBMAX];
int * aa3 = new int[PCBMAX];

float x[TOTALCOMPMAX]; //Temporary x panel coords for placement sequence.
float y[TOTALCOMPMAX];  //Temporary y panel coords for placement sequence.
float sx[TOTALSLOTMAX]; //Temporary x panel coords for slot locations.
float sy[TOTALSLOTMAX]; //Temporary y panel coords for slot locations.

float PCBw[PCBMAX];       //(pointers to the) width array
float PCBh[PCBMAX];       //(pointers to the) width array

FILE *infile; //Input name

ofstream fout;                  //Output function for results file
ofstream fxout;                 //Output function for Excel file
// ofstream fdump;//
ofstream acadout;               //Acad output file

//Here come the input structures
```

```
struct components               //Component structure
{
  int ct;                       //Component type
  int pcbid;                    //PCB identification type
  float xloc;                   //LOCAL (PCB-relative) x-location (was double)
  float yloc;                   //LOCAL (PCB-relative) y-location(was double)
};

struct feeder                   //feeder (rack) structure
{
  int ct;                       //Component type assigned
  float xglob;                  //GLOBAL x-location(was double)
  float yglob;                  //GLOBAL y-location(was double)
};
struct pattern                  //pattern structure
{
  float XX[PCBMAX];             //PCB center GLOBAL x-location(was double)
  float YY[PCBMAX];             //PCB center GLOBAL y-location(was double)
  int OO[PCBMAX];               //PCB center GLOBAL orientation (degrees)
};

//NOTE that the following sets are POINTERS, not structure in themselves

components * comp = new components[TOTALCOMPMAX];//component structure
feeder * slot = new feeder[TOTALSLOTMAX];//feeder structure
pattern * patt = new pattern[PATTERNMAX];//pattern structure

//Initialize some of the program parts
void Initialize(void)
{
  int i,j,k,kk;
  for(k=0; k<TOTALSLOTMAX; k++)
    for(i=0; i<pcbnum; i++)  {
      xsum[i][k] = 0;
      ysum[i][k] = 0;
    }
  for(i=0; i<pcbnum; i++) pcb_rot_best[i] = pcb_rot_worst[i] = 0;

  pattbest = 0;
  pattworst = 0;
  magbest = 999999.;
  magworst = 0.;
}

/*************************************************************
// Write the centers to an AutoCad Script file for BEST case
// This script can be run on top of another drawing,
// overlaying the panel design with a set of component
// populations centeriods.
*************************************************************/
void WriteAcad( void){
        float score = 0;
        int i,j,k,kk;
        float offset = 250.0;           //Offset between views


//Write out the files which made up these calculations
acadout << "TEXT\n"
                <<  10 << "," << 160.0 << "\n4.0\n0\n"; //AIM
        acadout <<  pattfile << " " << pcbfile << " " << locations << "\n";

//Write out the component spread estimate score(s)
//This value is dependent upon whether WriteAcad() was called
//for the BEST case in compare() or the WORST
```

```
acadout << "TEXT\n"
                <<  10 << "," << 155.0 << "\n4.0\n0\n";
        acadout <<  "Estimate score:  " << magnitude << "\n";

//Write out the pattern and rotation for this solution
acadout << "TEXT\n"
                <<  10 << "," << 165.0 << "\n4.0\n0\n";
acadout << "Pattern " << kkk << ":  ( ";
for(i=0; i<pcbnum; i++){
        acadout <<  pcbrot[i];
if(i<pcbnum-1) acadout << ", ";
}
acadout << " )\n";

        //Set linetype back to continous for rest of work
        acadout  << "LINETYPE\nSET\nCONTINUOUS\n\n";

//Change color to red for centers
acadout << "COLOR\nRED\n";
        //Print a circle for each small center of PCB
kk=0;
for(i=0; i<pcbnum; i++){
  for(k=0; k<racknum; k++){
            acadout <<  "CIRCLE\n";
            acadout << comx[i][k];
            acadout << "," << comy[i][k];
            acadout <<  "\n" << 4.0 << "\n";
    acadout << "PLINE\n" << comx[i][k]-4 << "," << comy[i][k] << "\n";
    acadout << "A\nD\n90\n" << comx[i][k] << "," << comy[i][k]+4 << "\n";
    acadout << "L\n" << comx[i][k] << "," << comy[i][k] << "\n";
    acadout << comx[i][k]+4 << "," << comy[i][k] << "\n";
    acadout << "A\nD\n-90\n";
    acadout << comx[i][k] << "," << comy[i][k]-4 << "\n";
    acadout << "L\n" << comx[i][k] << "," << comy[i][k] << "\nC\n";
    acadout << "HATCH\nSOLID\nLAST\n\n";
    kk++;
         }
}
//Change color to blue for panel center
acadout << "COLOR\nBLUE\n";
//Then one black center for entire panel
        acadout <<  "CIRCLE\n";
        acadout << pnlcntr[0] << "," << pnlcntr[1];
        acadout <<  "\n" << 5.0 << "\n";
acadout << "PLINE\n" << pnlcntr[0]-5 << ","
         << pnlcntr[1] << "\n";
acadout << "A\nD\n90\n" << pnlcntr[0] << ","
         << pnlcntr[1]+5 << "\n";
acadout << "L\n" << pnlcntr[0] << ","
         << pnlcntr[1] << "\n";
acadout << pnlcntr[0]+5 << "," << pnlcntr[1] << "\n";
acadout << "A\nD\n-90\n";
acadout << pnlcntr[0] << "," << pnlcntr[1]-5 << "\n";
acadout << "L\n" << pnlcntr[0] << ","
         << pnlcntr[1] << "\nC\n";
acadout << "HATCH\nSOLID\nLAST\n\n";

if(papm){
//Since assign[] was important to the magnitude value,
//draw lines from each centroid to the assigned slot.
for(i=0; i<pcbnum; i++){
  for(j=0; j<racknum; j++){
          // Print LINE and vertices for "non-required" moves
            acadout <<  "LINE\n";
```

```
            acadout  << comx[i][j]  << ",";
            acadout  << comy[i][j]  << "\n";
            acadout  << slot[assign[j]].xglob << ",";
            acadout  << slot[assign[j]].yglob << "\n\n";
  }
}
}
        //Print from small centers to panel centers
else{
  kk=0;
  for(i=0; i<pcbnum; i++){
    for(k=0; k<racknum; k++){
              acadout <<  "LINE\n";
              acadout << comx[i][k];
              acadout << "," << comy[i][k] << "\n";
      acadout << pnlcntr[0] << "," << pnlcntr[1] << "\n\n";
      kk++;
            }
  }
}

//Change color back to black ("white" against a black background)
acadout << "COLOR\nWHITE\n";

acadout.flush();

}


//Write only end results (done for either big or small dump)
void smallWrite(void)
{
  int i, j, kk;
  fout.flush();
  fout << "Enumerations:\t" << stringer+1 << "\n";

  //Best results
  fout << "Best pattern:" << pattbest << "\n";
  for(i=0; i<pcbnum; i++) fout << pcb_rot_best[i] << "\t";
  fout << "\n";
  fout << magbest << "\n";
  fout << "bestpanelcntr:\t";
    for(i=0; i<2; i++) fout << pnlcntr_best[i] << "\t";
  fout << "\n";
  fout.flush();

  kk=0;
  for(i=0; i<pcbnum; i++){
    fout << "bestpcb# " << i << "\n";
    for(j=0; j<racknum; j++){
      fout << "comptype# " << j << "\t( " << cntrs_bestx[i][j]
<< ", " << cntrs_besty[i][j] << " ) " << "\n";
      kk++;
    }
  }
  fout << "\n";
  fout.flush();

  //Worst results
  fout << "Worst pattern:" << pattworst << "\n";
  for(i=0; i<pcbnum; i++) fout << pcb_rot_worst[i] << "\t";
  fout << "\n";
  fout << magworst << "\n";
  fout << "worstpanelcntr:\t";
```

```
    for(i=0; i<2; i++) fout << pnlcntr_worst[i] << "\t";
  fout << "\n";
  fout.flush();

  kk=0;
  for(i=0; i<pcbnum; i++){
   fout << "worstpcb# " << i << "\n";
    fout.flush();
    for(j=0; j<racknum; j++){
      fout << "comptype# " << j << "\t( " << cntrs_worstx[i][j]
<< ", " << cntrs_worsty[i][j] << " ) " << "\n";
      kk++;
    }
  }
  fout << "\n";
  fout.flush();


}

//Write to a big file all enumerated results
void bigWrite(void)
{
  int i, j, kk;

  fout << stringer << "\t" << magnitude << "\t";
  for(i=0; i<pcbnum; i++) fout << pcbrot[i] << " " ;
  fout << "\n";

  for(i=0; i<2; i++) fout << pnlcntr[i] << "\t";
  fout << "\n";

  kk=0;
  for(i=0; i<pcbnum; i++){
    fout << "pcb# " << i << "\n";
    for(j=0; j<racknum; j++){
      fout << "comptype# " << j << "\t( " << comx[i][j]
<< ", " << comy[i][j] << " ) " << "\n";
      kk++;
    }
  }
  fout << "\n";
  fout.flush();

}

//Write to an Excel file all enumerated results
//All you get is pattern (elsewhere), enumeration #, magnitude, and pcbrot.
void excelWrite(void)
{
  int i;

  fxout << stringer << "\t" << magnitude << "\t";
  for(i=0; i<pcbnum; i++) fxout << pcbrot[i] << " " ;
  fxout << "\n";
  fxout.flush();

}

/*************************************************************************/
// Assigns index of panel output file name as a character
// Used for multiple output of ACAD script files
/*************************************************************************/
char *index1( int j ) {
```

```
        /* Up to INDEXMAX panel output files are writable
 */
        /* (Included 20 characters for possible expansion
 */
        static char *textnum[]=
        {
          "0",
          "1",
          "2",
          "3",
          "4",
          "5",
          "6",
          "7",
          "8",
          "9",
          "10",
          "11",
          "12",
          "13",
          "14",
          "15",
          "16",
          "17",
          "18",
          "19",
          "X"
        };

        if (j >= 0 && j <= 20 )
                return (textnum[j]);
        else
        {
                printf("\nNumber of index exceeds %d", 20);
                return (textnum[0]);
                exit(1);
        }
}

//  Set alpha vector from rotation requirements
void alphaSet( int kk, int alp ) {

        switch ( alp/90 )
        {

        // alp = 0 degrees
        case 0:
        aa0[kk] = 1;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // alp = 90 degrees
        case 1:
        aa0[kk] = 0;
        aa1[kk] = 0;
        aa2[kk] = 1;
        aa3[kk] = 0;
        break;

        // alp = 180 degrees
        case 2:
        aa0[kk] = 0;
```

```
        aa1[kk] = 1;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // alp = 270 degrees

        case 3:
        aa0[kk] = 0;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 1;
        break;

        // alp = 360 degrees
        case 4:
        aa0[kk] = 1;
        aa1[kk] = 0;
        aa2[kk] = 0;
        aa3[kk] = 0;
        break;

        // fall-out
        default:
        cout << "\nPCB " << kk << ", degree = " << alp << "\n";
        cout << "aa0["<<kk<<"]= "<<aa0[kk]<<  ", aa1["<<kk<<"]= "<<aa1[kk]
          << ", aa2["<<kk<<"]= "<<aa2[kk]<<  ", aa3["<<kk<<"]= "<<aa3[kk];
        cout << "\n\nSomething wrong with aa[] switching statement\n\n";
        exit(1);
        }
}

//Setting up an initial allocation for papm based upon largest candidate
//Assumes all the slots are already organized from left to right, single file
//Does allow for unique PCB in the same panel, but only one feeder row.
//The latter would have to be modified if multiple banks (opposite/
// perpendicular sides) are part of the problem
void l_candidate(void)
{
  int a,b,i,j,k;
  int flipper;
  int value1, value2;
  int ct_total[racknum]; //Temporary holder to count comptypes
  int temp_order[racknum]; //Temporary holder for the order of comptypes

  //If racknum < 3, no need to bother with any of the complicated stuff

  if(racknum > 3) {

  for (i = 0; i < racknum; i++) {
    ct_total[i] = 0;
    temp_order[i] = i;
  }

  for (i = 0; i < racknum; i++) {
    for (j = 0; j < pcbnum; j++) {
ct_total[i] += typecount[j][i];
    }
  }

  //Now sort in order of greatest to least in temp_order[]
  for (i = 0; i < racknum-1; i++) {
    for (j = i; j < racknum; j++) {
if(ct_total[i]<ct_total[j]){
```

```
//Swap the tallies of the component types
value1 = ct_total[i];
value2 = ct_total[j];
ct_total[j] = value1;
ct_total[i] = value2;
//Swap the component types order
value1 = temp_order[i];
value2 = temp_order[j];
temp_order[j] = value1;
temp_order[i] = value2;
}
    }
  }

  //Select the center (or next-to-center) position
  if(racknum % 2 == 0) a = racknum/2;
  else a = (racknum+1)/2;

  for(j=0; j<a; j++){                          //Repeat til a node hits head
    for(i=0; i<racknum; i++){
      value1 = temp_order[j];
      value2 = temp_order[i];

      temp_order[i]  = value1;
      temp_order[j]  = value2;
    }                                          //and iterate through to end
  }

  //Now, temp_order is rotated such that the middle values are at front, then
  //largest values start after position a.  Invert this string and it is done

  //b is simply zero at first, but increases as you invert the front half
  b = 0;
  if( a%2 != 0 ) flipper = (a+1)/2 ;  //If odd, swap with a hole in middle
    else flipper = a/2 ;                    //If diff is even, o.k.

  for(i=0; i<flipper; i++, b++, a--){
      value1 = temp_order[a];
      value2 = temp_order[b];

      temp_order[b]  = value1;
      temp_order[a]  = value2;
  }

  //Done with largest candidate; largest number of comptypes are assigned around the middle
  //Put in assign[] and also in perm_assign for rejuvenation between iterations
  for (i = 0; i < racknum; i++) {
    assign[i] = temp_order[i];
    perm_assign[i] = temp_order[i];
  }

  }
  //Racknum is <4, so just assign in order and be done with it
  else{
    for (i = 0; i < racknum; i++) {
      assign[i] = i;
      perm_assign[i] = i;
    }

  }

}
```

```
//Read locations for PCB and rack from input file
void readLocationsFile( void ) {
  int i,j,k;
  int compnum;                  //Components per pcb (varies)
//  int rackcheck = 0;            //A check on separate file inputs
  int kount = 0;                //Panel component counter START @ ZERO!
  int ctype; //Tracks the component type under consideration

  if ((infile = fopen(locations,"r"))==NULL)        //AIM
    {
      cout << "\nCannot open input file " << locations << "\n";
      exit(1);
    }

  for (j = 0; j < pcbnum; j++){
    fscanf(infile, "%d",&compnum);      //Read number of components for PCB
    pcbcomps[j] = compnum; //Record it

    if(kount+compnum > TOTALCOMPMAX) {  // Look ahead to see if max comps exceeded
       cout << "\nkount > TOTALCOMPMAX\n";
       fclose(infile);
       exit(1);
    }
    for (i = 0; i < compnum; i++) {
      fscanf(infile, "%f",&(comp[kount]).xloc);
      fscanf(infile, "%f",&(comp[kount]).yloc);
      if(!aim) fscanf(infile, "%d",&(comp[kount]).ct); //If papm or rthm, read comp type
      else (comp[kount]).ct = 0; //if aim, just dummy the .ct
      (comp[kount]).pcbid = j;
      kount++;
    }

  }                                   //End of file reading

  totalcomps = kount;               //Set totalcomps to TOTAL number of comps

  //NEXT:  RACK DATA (simply not done for AIM case, slot[] left uninitialized
    if(!aim){
      fscanf(infile, "%d",&racknum);  //Read number of component types for PCB
      for (i = 0; i < racknum; i++) {
        fscanf(infile, "%f",&(slot[i]).xglob);
        fscanf(infile, "%f",&(slot[i]).yglob);
        (slot[i]).ct = i;                       // Just for initialization
assign[i] = i;
      }

    }
    else racknum = 1; //AIM has only one comptype

  //If not aim, then count the component types in each PCB
  k=0;
  if(!aim){
    //First initialize the counters
    for(i=0; i<pcbnum; i++) for(j=0; j<racknum; j++)
        typecount[i][j] = 0;
    for(i=0; i<pcbnum; i++) {
      for(j=0; j<pcbcomps[i]; j++){
        ctype = comp[k].ct;
        typecount[i][ctype]++;
        k++;
      }
    }
  }
  //NOTE:  Need to calculate the slotspace (i.e., distance between slots)
```

```
  //Assumes that there is the same distance between any and all slots
  slotspace = fabs(slot[0].xglob - slot[1].xglob);

  //Establish the component type (temporary) allocations
  if(!aim) l_candidate(); //initialize a component assignment where appropriate

  fclose(infile);

}

//This subroutine reads pcb component and rack input data
void readPCBFile( void) {
  int j;
  int fixer = 0;                        //A sum to check on no. of fixed PCB

  if ((infile = fopen(pcbfile,"r"))==NULL)
    {
      cout << "\nCannot open input file " << pcbfile << "!\n";
      exit(1);
    }

  fscanf(infile, "%d",&pcbnum);
  if(pcbnum > PCBMAX) {
      cout << "\npcbnum > PCBMAX\n";
      fclose(infile);
      exit(1);
      }

  for (j = 0; j < pcbnum; j++){
    fscanf(infile, "%d", &sqrect[j] );//Read shape types for each PCB
    fscanf(infile, "%f", &PCBw[j] );//Read PCB width
    fscanf(infile, "%f", &PCBh[j] );//Read PCB height
  }

  //Check width vs. ht of pcb and set pcb sqrect[]
  //Note that this will override any fixed PCB in a pattern
  for (j = 0; j < pcbnum; j++){
    if(PCBw[j]==PCBh[j]) sqrect[j] = 2;         //Square
    else sqrect[j] = 1;                         //Rectangle
  }

      fclose(infile);
}


//This subroutine reads pattern input data
void readPatternFile(void) {
  int i,j,pattcount,dummy;

    if ((infile = fopen(pattfile,"r"))==NULL)
        {
          cout << "\nCannot open " << pattfile << " !\n";
          exit(1);
        }

  fscanf(infile, "%d",&pattnum);
  if(pattnum > PATTERNMAX) {            //Check on maximum no. of patterns
      cout << "\npattnum > Maximum number of Patterns\n";
      fclose(infile);
      exit(1);
      }

  fscanf(infile, "%d",&dummy);
  if(pcbnum != dummy) {          //Check that pcbnum in patterns.txt matches
```

```
                                    //pcbnum in pcbV.txt
  cout << "\npcbnum from " << pcbfile << " != pcbnum in " << pattnum << "!\n";
    fclose(infile);
    exit(1);
  }

  //Read in all patterns
  for (j = 0; j < pattnum; j++){

    fscanf(infile, "%d",&pattcount);    //Read id of pattern (a dummy)
                                        //pattcount is just in file for readability
    for (i = 0; i < pcbnum; i++) {
      fscanf(infile, "%f",&(patt[j]).XX[i]);
      fscanf(infile, "%f",&(patt[j]).YY[i]);
      fscanf(infile, "%d",&(patt[j]).OO[i]);
    }
  }                                     //End of file reading

  fclose(infile);

}


void pushit (void){
  int test;
  stringer = 1;
  cout<<"\nEnter 1 for << by one\nEnter 2 for >> by one\nEnter 0 to leave\n";
  cin >> test;
  while(test!=0){
    if(test == 1) stringer = stringer << 1;
    else
      if(test==2) stringer = stringer >> 1;
    cout << "stringer = " << stringer;
    cout<<"\nEnter 1 for << by one\nEnter 2 for >> by one\nEnter 0 to leave\n";
    cin >> test;
  }
}


void enumerator (void){
  unsigned long c = stringer;
  for(int i=0; i<sizer; i++) {
    cout << "\nvalue: " << c << "\n";
    c--;
  }
}

void indexmaker ( void){
  int i, test;
  for(i=0; i<pcbnum; i++){
    cout << "\npcbrot[" << i << "]: ?  ";cout.flush();
    cin >> pcbrot[i];
  }
  stringer = getindex();
  cout << "\nThe pcb set index is at " << stringer << "\n";
  test = int (stringer);
  cout << "\nAnd if converted to simple int, index is " << test << "\n";
}

//changes all the pcbrot[] values to that for a single integer input
void rotator (void){
  int i, test;
  for(i=0; i<pcbnum; i++){
    pcbrot[i] = getrot(i);
```

```
  }
}

//Returns a single index value for the pcbrot[] values, all together
unsigned long getindex (void){
  int i,j;
  unsigned long dummy = 0;
  stringer = 0;
  for(i=0; i<pcbnum; i++){
    dummy = (unsigned long)pcbrot[i];
    if(i>0){                                  //Don't push out for 0
      for(j=1; j<=i; j++) dummy = dummy << sqrect[j-1];
    }
    stringer = stringer + dummy;              //Tack the two together
  }
  return stringer;
}

//Changes a single pcbrot[pcb] value based upon the pcb passed
//and the current stringer value
int getrot (int pcb) {
  unsigned long c;
  int rhs = 0;                  //Right Hand Side
  int lhs = 0;                  //Left Hand Side
  c = stringer;
  int total = 32;               //Total number of +digits in a long (signed)

  if(sqrect[pcb]==0) return 0; //If sqrect for pcb of interest is fixed,
                               //then only one rotation possible:  zero degrees.
  if(pcb>0)  {                 //If not 0 pcb, figure out how many rhs digits
    for(int k=0; k<pcb; k++)   //Digits to right of pcb marker of interest
      rhs = rhs + sqrect[k];
  }
  lhs =total-rhs-sqrect[pcb];  //Digits to left of pcb marker of interest
  c = c << lhs;                //Eliminate digits to left of pcb marker
  c = c >> (lhs+rhs);          //Eliminate digits to right of pcb marker
  return int(c);
}

//Swaps two indices in assign[]
void swap(int a, int b)
{
int value1 = assign[a];
int value2 = assign[b];

assign[b]  = value1;
assign[a]  = value2;
}

//Compare current magnitude with previous and store
void compare(void)
{
  int i, j, kk;

  kk = 0;
  //Best results go here...
  if(magnitude < magbest) {
    magbest = magnitude;
    pattbest = kkk;
    for(i=0; i<2; i++) pnlcntr_best[i] = pnlcntr[i];
    for(i=0; i<pcbnum; i++){
      pcb_rot_best[i] = pcbrot[i]; //DOES NOT INCLUDE .OO!!!
    }
    for(i=0; i<pcbnum; i++){
```

```
      for(j=0; j<racknum; j++){
        cntrs_bestx[i][j] = comx[i][j]; //Scrolls through each PCB
        cntrs_besty[i][j] = comy[i][j]; //and each comp type within each
        kk++;
      }
    }
    if(acadbest=='y') {
      acadout.open(acadfilebest);
      WriteAcad();
      acadout.close();
    }

  }

  //And worst results go here...
  if(magnitude > magworst) {
    magworst = magnitude;
    pattworst = kkk;
    for(i=0; i<2; i++) pnlcntr_worst[i] = pnlcntr[i];
    for(i=0; i<pcbnum; i++){
      pcb_rot_worst[i] = pcbrot[i]; //DOES NOT INCLUDE .OO!!!
    }
    for(i=0; i<pcbnum; i++){
      for(j=0; j<racknum; j++){
        cntrs_worstx[i][j] = comx[i][j]; //Scrolls through each PCB
        cntrs_worsty[i][j] = comy[i][j]; //and each comp type within each
        kk++;
      }
    }
    if(acadworst=='y') {
      acadout.open(acadfileworst);
      WriteAcad();
      acadout.close();
    }
  }
}

//PAPM cases need to first calculate the PCB component type centroids,
//then use pairwise exchanges to get a good magnitude for
//the pattern and rotation set under consideration
//This subroutine calculates the magnitude only
float getPAPMmag(void)
{
  int i,j;
  float score = 0;

  for(i=0; i<pcbnum; i++){
    for(j=0; j<racknum; j++){
      score += sqrt( pow( comx[i][j]- slot[assign[j]].xglob, 2 )
                   + pow( comy[i][j]- slot[assign[j]].yglob, 2 ) )
     * float(typecount[i][j]) / float(totalcomps);
    }
  }

  return score;
}

//This is the simple pairwise exchange used to improve the papm estimate score.
void pwxchange(void)
{
  int i,j;
  int flag = 0;
  float newmag;
```

```
  magnitude = getPAPMmag();
  //Do this until no further improvements work out
  do{
    flag=0;
    for(i=0; i<racknum-1; i++){
      for(j=i+1; j<racknum; j++){
        swap(i,j);
        newmag = getPAPMmag();
        if( newmag < magnitude ){
  magnitude = newmag;
  flag = 1;
        }
//If no improvement, just switch back indices; magnitude is unchanged
else swap(j,i);
      }
    }
  }while(flag);
}

//Calculate estimates for papm
//Note that component types are handled via the racknum
//Also note that the global kkk sets the pattern number
//Does a simple pairwise exchange to get an approximate,
//minimum estimate
void papmestimator(void)
{
  int i,j,k;

  //Reset the allocation for this PCB rotation set
  for(i=0; i<racknum; i++) assign[i] = perm_assign[i];

  pnlcntr[0] = pnlcntr[1] = 0.; //Initialize these
  magnitude = 0;

  //Set the aa matrix, accounting for possible .OO != 0
  for(i=0; i<pcbnum; i++){
    if(sqrect[i] == 0) alphaSet( i, patt[kkk].OO[i] );
    else if(sqrect[i] == 1) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*180 );
    else if(sqrect[i] == 2) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*90 );
    for(j=0; j<racknum; j++) comx[i][j]= comy[i][j]= 0.;//Initialize these
  }

  //Assumes that each pcb has at least one of each comptype in it.
    for (i=0; i<pcbnum; i++) {
      for (j=0; j<racknum; j++) {
        comx[i][j]= patt[kkk].XX[i]
 + xsum[i][j] * aa0[i]
 - xsum[i][j] * aa1[i]
 - ysum[i][j] * aa2[i]
 + ysum[i][j] * aa3[i] ;

        comy[i][j]= patt[kkk].YY[i]
 + ysum[i][j] * aa0[i]
 - ysum[i][j] * aa1[i]
 + xsum[i][j] * aa2[i]
 - xsum[i][j] * aa3[i] ;

      pnlcntr[0] += comx[i][j]*float(typecount[i][j])/float(totalcomps);
      pnlcntr[1] += comy[i][j]*float(typecount[i][j])/float(totalcomps);
      }
    }

  //Running the pwxchange will get the magnitude set
  pwxchange();
```

```
}

//Calculate estimates for rthm
//Note that component types are accounted for in this case
//Also note that the global kkk sets the pattern number
void rthmestimator(void)
{
  int i,j,k;
  pnlcntr[0] = pnlcntr[1] = 0.;                 //Initialize these
  magnitude = 0;

  //Set the aa matrix, accounting for possible .OO != 0
  for(i=0; i<pcbnum; i++){
    if(sqrect[i] == 0) alphaSet( i, patt[kkk].OO[i] );
    else if(sqrect[i] == 1) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*180 );
    else if(sqrect[i] == 2) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*90 );
    for(j=0; j<racknum; j++) comx[i][j]= comy[i][j]= 0.;//Initialize these
  }

  //Assumes that each pcb has at least one of each comptype in it.
    for (i=0; i<pcbnum; i++) {
      for (j=0; j<racknum; j++) {
        comx[i][j]= patt[kkk].XX[i]
         + xsum[i][j] * aa0[i]
         - xsum[i][j] * aa1[i]
         - ysum[i][j] * aa2[i]
         + ysum[i][j] * aa3[i] ;

        comy[i][j]= patt[kkk].YY[i]
         + ysum[i][j] * aa0[i]
         - ysum[i][j] * aa1[i]
         + xsum[i][j] * aa2[i]
         - xsum[i][j] * aa3[i] ;

      pnlcntr[0] += comx[i][j]*float(typecount[i][j])/float(totalcomps);
      pnlcntr[1] += comy[i][j]*float(typecount[i][j])/float(totalcomps);
      }
    }

  //Now, find distances of each sub group centroid from panel centroid
  //Use sum of the magnitudes to estimate the spread of the population.
  //If all PCB are identical, then the typecount/totalcomps will cancel out
  for(i=0; i<pcbnum; i++) {
    for (j=0; j<racknum; j++) {
      magnitude += sqrt( pow( comx[i][j]- pnlcntr[0], 2 )
                    + pow( comy[i][j]- pnlcntr[1], 2 ) )
     * float(typecount[i][j]) / float(totalcomps);
    }
  }
}

//Calculate estimates for aim
//Note that component types are not an issue for aim
//Also note that the global kkk sets the pattern number
void aimestimator(void)
{
  int i,j;

  pnlcntr[0] = pnlcntr[1] = 0.; //Initialize these
  magnitude = 0;

  //Set the aa matrix, accounting for possible .OO != 0
  for(i=0; i<pcbnum; i++){
    if(sqrect[i] == 0) alphaSet( i, patt[kkk].OO[i] );
```

```
    else if(sqrect[i] == 1) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*180 );
    else if(sqrect[i] == 2) alphaSet( i, patt[kkk].OO[i] + pcbrot[i]*90 );
    for(j=0; j<racknum; j++) comx[i][j]= comy[i][j]= 0.;//Initialize these
  }

  //Calculate the total panel "centroid", which is simply a combination
  //of the smaller centroids for the particular aa[] set
    for (i=0; i<pcbnum; i++) {
      comx[i][0]= patt[kkk].XX[i]
   + xsum[i][0] * aa0[i]
   - xsum[i][0] * aa1[i]
   - ysum[i][0] * aa2[i]
   + ysum[i][0] * aa3[i] ;

      comy[i][0]= patt[kkk].YY[i]
   + ysum[i][0] * aa0[i]
   - ysum[i][0] * aa1[i]
   + xsum[i][0] * aa2[i]
   - xsum[i][0] * aa3[i] ;

    pnlcntr[0] += comx[i][0]*pcbcomps[i]/float(totalcomps);
    pnlcntr[1] += comy[i][0]*pcbcomps[i]/float(totalcomps);
    }


  //Now, find distances of each sub group centroid from panel centroid
  //Use sum of the magnitudes to estimate the spread of the population.
  //If all PCB are identical, then the pcbcomps/totalcomps will cancel out
  for(i=0; i<pcbnum; i++) {
    magnitude += sqrt( pow( comx[i][0]- pnlcntr[0], 2 )
                     + pow( comy[i][0]- pnlcntr[1], 2 ) );
  }

}

void main (void){

  char machine;
  int checkmach = 0;
  time_t timer; //Used to catch time for files writing
  char * timepoint; //Used to catch time for files writing
  int ctype; //Tracks the component type under consideration
  int i,j,k, kk;

  //Prompt for filenames and machine type
    cout << "\nEnter one of the following:
               a for aim,
               p for papm,
               r for rthm, or
               q for quit.\n";
    cout.flush();
  cin >> machine;
  while(machine != 'q' && machine != 'a' && machine != 'p' && machine != 'r' )
  {
    cout << "\n" << machine << " is not a valid selection..." ;
    cout << "\nEnter one of the following:
               a for aim,
               p for papm,
               r for rthm, or
               q for quit.\n";
    cin >> machine;
  }

    if(machine == 'q') exit(1);
```

```
  if( machine == 'a') {
    aim  = 1;
    papm = 0;
    rthm = 0;
  }
  else if( machine == 'p') {
    aim  = 0;
    papm = 1;
    rthm = 0;
  }
  else if( machine == 'r') {
    aim  = 0;
    papm = 0;
    rthm = 1;
  }

//Get a good time hack
time( &timer);
timepoint = ctime( &timer );  //A time hack, used for file management
//Replace the blanks in the time hack with "_"
k=0;
do{
k++;
if(*(timepoint+k)==' ') *(timepoint+k) = '_';
}while(*(timepoint+k)!='\0' && k<30);
cout << timepoint <<"\n";
cout.flush();

//Set output file name
strcpy (outfile, name1);
if(aim) *(timepoint+k-1) = 'A';
else if(papm) *(timepoint+k-1) = 'P';
else if(rthm) *(timepoint+k-1) = 'R';
strcat(outfile, timepoint);
strcat(outfile, ".out");

cout << "\nEnter LOCATIONS index for filename(dA#.txt or dP#.txt), \n";
cout << "1 through 15:\n";
cout.flush();
cin >> lindex;
cout << lindex << "\n";

//Set locations file name
strcpy (locations, "d");
if(aim) strcat (locations, "A");
else if(papm || rthm) strcat (locations, "P");
strcat(locations, index1(lindex) );
strcat(locations, ".txt");

cout << "\nEnter number of PCB, from 2 to 8:\n";
cout.flush();
cin >> pcbnum;
cout << pcbnum << "\n";

//Set Acad output best filename
strcpy (acadfilebest, name0);
if(aim) strcat (acadfilebest, "Best_A");
else if(papm) strcat (acadfilebest, "Best_P");
else if(rthm) strcat (acadfilebest, "Best_R");
strcat(acadfilebest, index1(pcbnum) );
strcat(acadfilebest, "_d");
strcat(acadfilebest, index1(lindex) );
strcat(acadfilebest, ".scr");
```

```
//Set Acad output worst filename
strcpy (acadfileworst, name0);
if(aim) strcat (acadfileworst, "Worst_A");
else if(papm) strcat (acadfileworst, "Worst_P");
else if(rthm) strcat (acadfileworst, "Worst_R");
strcat(acadfileworst, index1(pcbnum) );
strcat(acadfileworst, "_d");
strcat(acadfileworst, index1(lindex) );
strcat(acadfileworst, ".scr");

cout << "\n\n" << pcbnum << " PCB will be used in this program.\n";
cout << "If you wish, the rest of the input can be automatic, based \n";
cout << "upon " << pcbnum << " PCB.  \n";
//Set output file name
strcpy (outfile, "Est_");
if(aim) strcat (outfile, "A");
else if(papm) strcat (outfile, "P");
else if(rthm) strcat (outfile, "R");
strcat(outfile, index1(pcbnum) );
strcat(outfile, "_d");
strcat(outfile, index1(lindex) );
strcat(outfile, ".txt");

//Set exfile file name
strcpy (exfile, "Est_");
if(aim) strcat (exfile, "A");
else if(papm) strcat (exfile, "P");
else if(rthm) strcat (exfile, "R");
strcat(exfile, index1(pcbnum) );
strcat(exfile, "_d");
strcat(exfile, index1(lindex) );
strcat(exfile, ".xls");

//Set pcb input file name
if(aim) strcpy (pcbfile, "aimpcbV");
else if(papm ) strcpy (pcbfile, "papmpcbV");
else if(rthm ) strcpy (pcbfile, "rthmpcbV");
strcat(pcbfile, index1(pcbnum) );
strcat(pcbfile, ".txt");
//Set patterns input file name
strcpy(pattfile, "patterns");
strcat(pattfile, index1(pcbnum) );
strcat(pattfile, ".txt");

cout << "\nInput pcb file will be " << pcbfile << ";\n";
cout << "Input locations file will be " << locations << ";\n";
cout << "Input pattern file will be " << pattfile << ";\n";
cout << "Output file will be " << outfile << ";\n\n";
cout << "Enter y for accept filenames, \n";
cout << "enter n to enter a different outfile name:\n\n";
cin >> prompt;
while(prompt != 'y' && prompt != 'n' && prompt !='q')
{
  cout << "\n" << prompt << " is not a valid selection..." ;
  cout << "\n\tEnter one of the following:\n";
  cout << "\t\t y for accepting default filenames, or\n";
  cout << "\t\t n for inputting new filenames, or\n";
  cout << "\tq to quit.\n";
  cin >> prompt;
}

if(machine == 'q') exit(1);
```

```
if(prompt == 'n'){
cout << "\nEnter PCB filename, <mach>pcbV#.txt :\n\n";
cout.flush();
cin >> pcbfile;
cout << pcbfile;

cout << "\nEnter PATTERN filename, either patterns#.txt or patternsV#.txt; ";
cout << "only the first pattern will be used:\n\n";
cin >> pattfile;
cout << pattfile;
cout.flush();

cout << "\nEnter Locations filename, dA#.txt or dP#.txt :\n";
cout.flush();
cin >> locations;
cout << locations;
}

cout << "\n\nDo you want an output file of ALL the rotation combination\n";
cout << " results?  Keep in mind that this can take up megabytes for\n";
cout << " problems of modest size (say, 8 square and ";
cout << " dissimilar PCB)...\n\n";
cout << "Enter y for yes or n for no:\n";
cout.flush();
cin >> bigdump;
while(bigdump!='y' && bigdump !='n'){
   cout << "\nEnter y for yes or n for no:\n";
cout.flush();
cin >> bigdump;
}

cout << "\n\nDo you want an output file of only the BEST and WORST\n";
cout << " results?  This option will print to screen and to a small file.\n";
cout << "Enter y for yes or n for no:\n";
cout.flush();
cin >> smalldump;
while(smalldump!='y' && smalldump !='n'){
   cout << "\nEnter y for yes or n for no:\n";
cout.flush();
cin >> smalldump;
}

cout << "\n\nDo you want an EXCEL-oriented output file of all the enumerations?\n";
cout << "This option may produce a file which is too big to be parsed by Excel.\n\n";
cout << "Enter y for yes or n for no:\n";
cout.flush();
cin >> exceldump;
while(exceldump!='y' && exceldump !='n'){
   cout << "\nEnter y for yes or n for no:\n";
cout.flush();
cin >> exceldump;
}


cout << "\n\nDo you want an AutoCad script of BEST results layout?\n\n";
cout << "Enter y for yes or n for no:\n";
cout.flush();
cin >> acadbest;
while(acadbest!='y' && acadbest !='n'){
   cout << "\nEnter y for yes or n for no:\n";
cout.flush();
cin >> acadbest;
}
```

```
  cout << "\n\nDo you want an AutoCad script of WORST results layout?\n";
  cout << " results?  This option will print to screen and to a small file.\n\n";
  cout << "Enter y for yes or n for no:\n";
  cout.flush();
  cin >> acadworst;
  while(acadworst!='y' && acadworst !='n'){
     cout << "\nEnter y for yes or n for no:\n";
  cout.flush();
  cin >> acadworst;
  }

  readPCBFile(); //Read in pcbfile information
  readPatternFile(); //Read in pattern information
  readLocationsFile(); //Read in location information


  for(i=0; i<pcbnum; i++) if(sqrect[i]!=0) sizer = sqrect[i]+sizer;
  sizer = (unsigned long)(pow(2,sizer));//Now the size is total no. of combos
  stringer = sizer-1;                   //Stringer includes 0th option, so -1 of
                                        //sizer
  Initialize();

  if(bigdump == 'y' || smalldump == 'y') {
    fout.open(outfile); //Open and leave open
    fout << pcbfile << "\t" << pattfile << "\t" << locations << "\n";
  }

  if(exceldump == 'y' ){
    fxout.open(exfile); //Open and leave open
    fxout << pcbfile << "\t" << pattfile << "\t" << locations << "\n";
  }

  //Run throught all patterns in file; if only one pattern,
  //the loop only runs once, of course.
  for(kkk = 0; kkk< pattnum; kkk++){

  cout << "\nPattern " << kkk << "\n";

  //Initialize these every new pattern
  for(i=0; i<pcbnum; i++) for(k=0; k<racknum; k++){
    xsum[i][k] = 0;
    ysum[i][k] = 0;
  }


  if(bigdump == 'y') {
    fout << "pattern " << kkk << "\n";
  }

  if(exceldump == 'y') {
    fxout << kkk << "\n";
  }

  //Calculate the PCB component population centers
  //For AIM, it is a single x,y pair, since there is only one component type
  //For PAPM and RTHM, there are as many x,y pairs as component types
  kk=0;
  for(i=0; i<pcbnum; i++){
    for(k=0; k<pcbcomps[i]; k++){
      if(aim){
xsum[i][0] += comp[kk].xloc / float(pcbcomps[i]);
ysum[i][0] += comp[kk].yloc / float(pcbcomps[i]);
      }
      //If papm or rthm...
```

```
      else {
  ctype = comp[kk].ct;
          xsum[i][ctype] += comp[kk].xloc/float(typecount[i][ctype]);
          ysum[i][ctype] += comp[kk].yloc/float(typecount[i][ctype]);
      }
    kk++;
    }
  }

  //Run through all enumerations of PCB rotations and compare
  for(j=0; j<sizer; j++){
    stringer = j; //Set the single array index value
    rotator(); //Set pcbrot[] for the pcb

    //Call appropriate adjustor for the machine type
    if(aim) aimestimator();
    else if(papm)  papmestimator();
    else if(rthm)  rthmestimator();
    else {cout << "Neither aim, papm, nor rthm\n" ; exit(1);}

    //Call a comparison and store best and worst
    compare();

    //Write the results to file (this is NOT a good idea for large problems
    if(bigdump=='y') bigWrite();

    //Write the results to Excel file (this is NOT a good idea for large problems
    if(exceldump=='y') excelWrite();
    }
  }

  if(smalldump=='y' || bigdump=='y') smallWrite();
  if(bigdump=='y' || smalldump=='y') fout.close();
  if(exceldump=='y') fxout.close();
  if(smalldump) cout << "Finder is finished\n";

  return 0;
}
```

Appendix D

Explanation of errors in Leu et. al. [48].

This appendix documents the verification of the error published in one of the articles used to check the `panelizer` program. The article is cited as: M. C. Leu, H. Wong, Z. Ji, "Planning of Component Placement/Insertion Sequence and Feeder Setup in PCB Assembly Using Genetic Algorithm," Transactions of the ASME, vol 115, Dec 1993, 424-432. The article was contributed by the Electrical and Electronic Packaging Division for publication in the Journal of Electronic Packaging as well [48].

The genetic algorithm was applied on the test panel as listed in Table 4.8. The head index was 0.25 seconds, the table fixture speed was 60 mm/s, the feeder motion speed was also 60 mm/s and there were 15 mm between each adjacent feeder slot location. Thus, the time required to move between adjacent slots was 0.25 seconds. With 50 components on the panel, the fastest possible placement time for the panel would be if the head was not required to stop for any component placement, resulting in 12.5 seconds.

The authors reported a placement time for the panel, using a Chebyshev metric, of 12.75 seconds, which was near-optimal. However, a figure was given which illustrated the placement sequence for the optimal placement. This sequence was checked via spreadsheet calculations and yielded a result of 50 seconds. The sequence of the component solution and the spreadsheet calculations are shown in figure D.1.

If an Euclidean metric is assumed for the same solution, the panel placement time would be 52.4 seconds. The spreadsheet calculations are given in D.2

There is a figure 18 in the article which is referenced as the GA convergence trend; this figure converges to a final value of approximately 52 seconds, which matches the Euclidean metric analysis.

Therefore, the Euclidean metric was used for the RTHM experiments in this dissertation. It is of interest to note that the placement sequence and allocation solution given in Leu et. al. was not especially good for either metric, since the maximum time for all the component moves was always the board time.

turret index	vel (mm/s)	fdr dst/slo	Feeder vel	feeder /slot						
0.25	60	15	60	0.25						
Comp	x	y	type	Feeder pos to 5	Feeder pos 6-10	Feeder time	board time	brd time <0.25?	place time	TOTAL
16	140	180	10	10	3	0.5	0.5	NO	0.5	50.25
5	100	230	4	7	8	1.25	0.8333	NO	1.25	
21	160	180	4	7	8	0	1	NO	1	
20	160	140	5	6	5	0.75	0.6667	NO	0.75	
45	240	120	1	2	2	0.75	1.3333	NO	1.333333	
33	200	180	10	10	3	0.25	1	NO	1	
47	240	180	7	7	8	1.25	0.6667	NO	1.25	
46	240	140	10	10	3	1.25	0.6667	NO	1.25	
31	200	140	3	10	3	0	0.6667	NO	0.666667	
37	220	100	10	10	3	0	0.6667	NO	0.666667	
41	240	40	10	10	3	0	1	NO	1	
28	200	60	8	8	9	1.5	0.6667	NO	1.5	
23	180	60	9	9	1	2	0.3333	NO	2	
7	120	90	10	10	3	0.5	1	NO	1	
2	100	90	3	10	3	0	0.3333	NO	0.333333	
13	140	80	2	4	4	0.25	0.6667	NO	0.666667	
17	140	220	7	7	8	1	2.3333	NO	2.333333	
10	120	190	5	6	5	0.75	0.5	NO	0.75	
11	120	230	9	9	1	1	0.6667	NO	1	
4	100	180	10	10	3	0.5	0.8333	NO	0.833333	
3	100	130	2	4	4	0.25	0.8333	NO	0.833333	
19	160	100	2	4	4	0	1	NO	1	
14	140	100	9	9	1	0.75	0.3333	NO	0.75	
18	160	60	5	6	5	1	0.6667	NO	1	
29	200	100	9	9	1	1	0.6667	NO	1	
36	220	60	9	9	1	0	0.6667	NO	0.666667	
24	180	100	5	6	5	1	0.6667	NO	1	
8	120	130	6	6	5	0	1	NO	1	
1	100	60	6	6	5	0	1.1667	NO	1.166667	
6	120	50	4	7	8	0.75	0.3333	NO	0.75	
12	140	40	9	9	1	1.75	0.3333	NO	1.75	
35	220	40	9	9	1	0	1.3333	NO	1.333333	
42	240	60	9	9	1	0	0.3333	NO	0.333333	
43	240	80	4	7	8	1.75	0.3333	NO	1.75	
44	240	100	8	8	9	0.25	0.3333	NO	0.333333	
30	200	130	9	9	1	2	0.6667	NO	2	
38	220	160	9	9	1	0	0.5	NO	0.5	
48	240	200	6	6	5	1	0.6667	NO	1	
40	220	220	5	6	5	0	0.3333	NO	0.333333	
50	240	220	2	4	4	0.25	0.3333	NO	0.333333	
39	220	200	7	7	8	1	0.3333	NO	1	
32	200	170	7	7	8	0	0.5	NO	0.5	
49	240	210	7	7	8	0	0.6667	NO	0.666667	
34	200	220	4	7	8	0	0.6667	NO	0.666667	
22	160	220	9	9	1	1.75	0.6667	NO	1.75	
27	180	220	8	8	9	2	0.3333	NO	2	
26	180	180	4	7	8	0.25	0.6667	NO	0.666667	
25	180	140	8	8	9	0.25	0.6667	NO	0.666667	
15	140	140	4	7	8	0.25	0.6667	NO	0.666667	
9	120	150	9	9	1	1.75	0.3333	NO	1.75	

Figure D.1: Spreadsheet calculations for Chebyshev solution by Leu et. al.

turret index	vel (mm/s)	fdr dst/slot	Feeder vel	feeder /slot						
0.25	60	15	60	0.25						
Comp	x	y	type	Feeder pos to 5	Feeder pos 6-10	Feeder time	board time	brd time <0.25?	place time	TOTAL
16	140	180	10	10	3	0.5	0.6009	NO	0.600925	52.37
5	100	230	4	7	8	1.25	1.0672	NO	1.25	
21	160	180	4	7	8	0	1.3017	NO	1.301708	
20	160	140	5	6	5	0.75	0.6667	NO	0.75	
45	240	120	1	2	2	0.75	1.3744	NO	1.374369	
33	200	180	10	10	3	0.25	1.2019	NO	1.20185	
47	240	180	7	7	8	1.25	0.6667	NO	1.25	
46	240	140	10	10	3	1.25	0.6667	NO	1.25	
31	200	140	3	10	3	0	0.6667	NO	0.666667	
37	220	100	10	10	3	0	0.7454	NO	0.745356	
41	240	40	10	10	3	0	1.0541	NO	1.054093	
28	200	60	8	8	9	1.5	0.7454	NO	1.5	
23	180	60	9	9	1	2	0.3333	NO	2	
7	120	90	10	10	3	0.5	1.118	NO	1.118034	
2	100	90	3	10	3	0	0.3333	NO	0.333333	
13	140	80	2	4	4	0.25	0.6872	NO	0.687184	
17	140	220	7	7	8	1	2.3333	NO	2.333333	
10	120	190	5	6	5	0.75	0.6009	NO	0.75	
11	120	230	9	9	1	1	0.6667	NO	1	
4	100	180	10	10	3	0.5	0.8975	NO	0.897527	
3	100	130	2	4	4	0.25	0.8333	NO	0.833333	
19	160	100	2	4	4	0	1.118	NO	1.118034	
14	140	100	9	9	1	0.75	0.3333	NO	0.75	
18	160	60	5	6	5	1	0.7454	NO	1	
29	200	100	9	9	1	1	0.9428	NO	1	
36	220	60	9	9	1	0	0.7454	NO	0.745356	
24	180	100	5	6	5	1	0.9428	NO	1	
8	120	130	6	6	5	0	1.118	NO	1.118034	
1	100	60	6	6	5	0	1.2134	NO	1.213352	
6	120	50	4	7	8	0.75	0.3727	NO	0.75	
12	140	40	9	9	1	1.75	0.3727	NO	1.75	
35	220	40	9	9	1	0	1.3333	NO	1.333333	
42	240	60	9	9	1	0	0.4714	NO	0.471405	
43	240	80	4	7	8	1.75	0.3333	NO	1.75	
44	240	100	8	8	9	0.25	0.3333	NO	0.333333	
30	200	130	9	9	1	2	0.8333	NO	2	
38	220	160	9	9	1	0	0.6009	NO	0.600925	
48	240	200	6	6	5	1	0.7454	NO	1	
40	220	220	5	6	5	0	0.4714	NO	0.471405	
50	240	220	2	4	4	0.25	0.3333	NO	0.333333	
39	220	200	7	7	8	1	0.4714	NO	1	
32	200	170	7	7	8	0	0.6009	NO	0.600925	
49	240	210	7	7	8	0	0.9428	NO	0.942809	
34	200	220	4	7	8	0	0.6872	NO	0.687184	
22	160	220	9	9	1	1.75	0.6667	NO	1.75	
27	180	220	8	8	9	2	0.3333	NO	2	
26	180	180	4	7	8	0.25	0.6667	NO	0.666667	
25	180	140	8	8	9	0.25	0.6667	NO	0.666667	
15	140	140	4	7	8	0.25	0.6667	NO	0.666667	
9	120	150	9	9	1	1.75	0.3727	NO	1.75	

Figure D.2: Spreadsheet calculations for Euclidean solution by Leu et. al.

Appendix E

Solution Data for Experiments 2 and 3

Solutions below are given as "per-panel" and not as "per-component". Solutions are given vertically in terms of each experiment order, according to machine type. All machine types have Experiment 2 data and hence below give results for case A, B and C. However, PAPM and RTHM results also include those results of Experiment 3, below the first three solution output listings for each machine type.

The output gives each link in the composite chromosome as a "list." List 0, list 1, list 2, and list 3 correspond to $\tilde{s}^{\rho}$, $\tilde{s}^{\theta}$, $\tilde{s}^{\sigma}$, and $\tilde{s}^{*}$, respectively (see section 3.4). If an AIM experiment is conducted, list 3 is printed, but is merely dummy output, since the AIM has no component types in its problem formulation.

Notes on reading pattern and PCB rotation set information: The pattern file `patterns#.txt` is used in `panelizer.C` for either traditional or global analysis. The input pattern file has a set of initial orientations for the PCB that are in a given defined pattern. The output of an analysis will give a PCB rotation set which is defined *with respect to the* `patterns#.txt` *PCB rotation set* initially defined for the problem. As an example: If a

global solution had a pattern 5 defined with initial PCB orientation of `(0, 180, 0, 0)`, and the output of the analysis yielded a pattern 5 with a PCB rotation set of `(90, 90, 0, 0)`, then the PCB actual orientations, relative to their component definitions in `dP#.txt`, would be `(90, 270, 0, 0)`. Since a traditional analysis does not manipulate the pattern or the PCB initial orientations within that pattern, their solutions will always contain zero values for the patterns and PCB rotation set output (lists 0 and 1). But it should be remembered that the traditional analysis was based upon a `patterns#.txt` file which had a single pattern and initial PCB orientations that were not necessarily all at zero; they were based upon the *estimator* pattern results.

```
AIM 2 PCB GLOBAL

498.627
list 0: 0
list 1: 180 0
list 2: 19 17 16 20 28 24 25 27 26 14 8 12 15 13 5 7 6 4 0 2 3 1 9 11
10 30 31 29 21 23 22 18
list 3: 0

511.965
list 0: 0
list 1: 0 0
list 2: 3 1 0 9 11 14 10 8 15 13 12 4 5 6 2 7 30 31 29 20 21 23 22 18
19 17 16 28 24 25 26 27
list 3: 0

505.255
list 0: 0
list 1: 180 180
list 2: 25 16 24 17 19 18 22 14 12 8 10 11 9 1 0 3 2 6 7 4 5 13 15 23
21 20 28 29 31 30 26 27
list 3: 0

AIM 2 PCB TRAD BEST

498.627
list 0: 0
list 1: 0 0
list 2: 21 29 31 30 6 4 7 5 13 15 14 12 8 10 11 9 0 1 3 2 26 27 25 17
19 18 16 24 28 20 22 23
list 3: 0

499.639
list 0: 0
list 1: 0 0
list 2: 8 12 4 6 2 0 9 1 3 30 28 20 16 24 26 27 25 17 19 18 22 23 21
29 31 7 5 13 15 14 10 11
list 3: 0

499.135
list 0: 0
list 1: 0 0
list 2: 17 19 18 16 20 22 23 21 29 28 31 30 7 6 4 5 12 13 15 14 10 11
```

```
9 8 0 1 3 2 27 26 24 25
list 3: 0

AIM 2 PCB TRAD WORST

498.627
list 0: 0
list 1: 0 0
list 2: 21 29 31 30 6 4 7 5 13 15 14 12 8 10 11 9 0 1 3 2 26 27 25 17
19 18 16 24 28 20 22 23
list 3: 0

729.955
list 0: 0
list 1: 0 0
list 2: 26 27 25 24 16 17 19 18 22 23 21 20 28 29 31 15 13 5 7 6 2 4
12 8 0 3 1 9 11 10 14 30
list 3: 0

965.048
list 0: 0
list 1: 0 0
list 2: 13 5 4 7 6 0 2 3 1 9 11 10 8 12 14 15 31 29 21 23 22 18 19 17
25 27 26 24 16 20 28 30
list 3: 0

AIM 4 PCB GLOBAL

1085.93
list 0: 0
list 1: 180 270 0 90
list 2: 19 15 13 7 5 4 12 14 10 8 3 6 2 1 0 9 11 42 43 41 40 44 46 47
45 39 37 36 38 32 34 35 33 62 63 53 61 60 54 55 52 58 59 56 57 49 51
48 50 27 25 26 30 31 28 29 21 23 22 20 24 17 16 18
list 3: 0

765.414
list 0: 0
list 1: 0 270 90 180
list 2: 1 0 6 2 3 9 10 8 11 18 19 17 25 24 16 22 23 21 20 29 31 28 30
26 27 55 53 61 63 62 60 52 54 50 51 48 56 58 59 57 49 38 36 44 47 46
45 37 39 34 32 35 33 42 40 41 43 7 5 13 15 12 14 4
list 3: 0

506.28
list 0: 0
list 1: 0 270 90 180
list 2: 45 44 40 37 36 32 34 38 57 39 59 56 58 48 49 50 25 54 52 60 62
63 61 53 55 27 24 28 26 30 31 29 1 18 3 35 19 51 17 16 20 22 9 21 23
11 4 5 13 15 14 10 8 12 0 2 33 6 41 7 43 42 46 47
list 3: 0

AIM 4 PCB BEST TRADITIONAL

1080.74
list 0: 0
list 1: 0 0 0 0
list 2: 3 21 31 29 20 28 30 26 27 25 24 16 19 17 18 22 23 58 59 57 56
60 52 55 54 48 49 51 50 53 62 61 63 33 35 42 43 41 40 32 44 36 34 38
39 37 45 47 46 7 5 13 15 12 14 11 9 1 0 10 8 4 6 2
list 3: 0

743.524
list 0: 0
```

```
list 1: 0 0 0 0
list 2: 38 39 37 45 47 46 42 43 44 36 40 41 7 5 4 13 15 14 10 11 8 12
0 6 2 3 1 9 18 19 16 22 23 20 29 21 31 30 28 24 26 27 25 17 54 52 55
53 61 63 62 60 56 59 58 57 49 48 50 51 35 33 32 34
list 3: 0

453.732
list 0: 0
list 1: 0 0 0 0
list 2: 51 19 50 17 48 52 54 25 55 27 53 61 63 62 60 56 34 49 38 57 39
58 59 37 36 45 47 44 46 42 43 7 5 6 41 40 32 33 2 0 4 12 13 15 14 10 8
9 22 23 11 21 20 28 29 31 30 26 24 16 18 1 3 35
list 3: 0

AIM 4 PCB WORST TRADITIONAL

1080.74
list 0: 0
list 1: 0 0 0 0
list 2: 3 21 31 29 20 28 30 26 27 25 24 16 19 17 18 22 23 58 59 57 56
60 52 55 54 48 49 51 50 53 62 61 63 33 35 42 43 41 40 32 44 36 34 38
39 37 45 47 46 7 5 13 15 12 14 11 9 1 0 10 8 4 6 2
list 3: 0

1361.71
list 0: 0
list 1: 0 0 0 0
list 2: 25 27 26 30 28 29 31 13 4 0 1 3 2 6 7 5 12 11 9 8 10 14 15 47
46 45 44 36 39 37 38 34 33 35 41 32 40 42 43 55 54 50 51 57 49 59 58
56 48 52 53 60 62 61 63 21 23 22 18 16 20 19 17 24
list 3: 0

1533.43
list 0: 0
list 1: 0 0 0 0
list 2: 31 29 30 26 24 28 21 23 20 22 18 19 16 17 25 27 7 6 2 3 1 0 4
5 13 12 15 8 11 9 10 14 45 44 36 37 39 38 34 35 32 33 41 43 40 42 46
47 63 61 53 55 54 52 60 62 48 50 51 49 57 56 59 58
list 3: 0

AIM 8 PCB GLOBAL

2232.9595
list 0: 6
list 1: 0 0 180 180 0 0 180 180
list 2: 54 59 57 60 63 61 53 55 52 51 43 33 35 34 36 44 46 45 47 42 40
32 41 37 7 6 3 11 9 1 75 73 65 67 94 95 92 91 18 20 21 30 26 16 17 19
25 27 24 23 22 28 31 29 10 15 14 13 12 8 5 4 0 2 38 39 76 78 74 72 64
70 68 79 77 69 71 66 90 88 89 81 83 86 85 87 80 82 84 93 114 118 119
117 127 126 123 122 125 116 124 112 113 121 120 115 101 111 109 110
106 104 97 99 105 107 108 96 100 103 98 102 58 62 56 50 48 49
list 3: 0

1950.82
list 0: 4
list 1: 0 0 180 180 0 0 180 180
list 2: 82 89 90 91 66 67 69 71 77 79 78 76 72 75 74 73 65 64 68 70 85
93 95 94 92 83 81 107 80 88 84 106 96 111 110 105 104 97 98 99 38 39
37 2 1 0 9 8 11 15 13 12 10 14 4 7 5 6 3 31 23 20 17 25 18 22 21 29 19
16 28 30 24 26 27 35 40 42 47 45 43 48 52 56 59 57 62 58 63 60 61 53
55 54 50 49 51 46 36 44 32 34 33 41 102 103 100 101 109 108 118 119
116 117 125 127 123 122 126 124 120 112 121 113 114 115 87 86
list 3: 0
```

```
1652.54
list 0: 8
list 1: 0 180 0 0 0 180 180 180
list 2: 79 61 60 59 58 62 63 53 55 54 48 49 56 57 52 50 51 45 36 37 32
43 2 0 6 7 5 13 15 4 12 14 10 8 27 17 25 26 30 31 28 29 20 21 22 23 16
19 18 24 11 9 1 3 47 46 44 42 41 40 33 34 67 35 65 39 75 74 72 38 73
64 77 76 78 69 71 66 68 70 93 92 94 90 99 97 96 100 106 107 105 104
108 98 110 111 109 101 103 102 95 85 84 80 113 124 125 117 127 126 123
122 121 120 112 114 81 118 89 88 91 119 116 115 83 82 86 87
list 3: 0

AIM 8 PCB BEST TRADITIONAL

2105.79
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 12 13 5 6 7 4 1 8 9 0 3 2 35 33 41 32 38 34 36 47 45 46 42 43
40 44 37 39 106 110 108 96 100 99 97 107 104 105 98 102 103 101 109
111 57 59 58 56 62 61 63 60 48 49 51 50 52 55 53 54 79 78 72 77 76 69
66 95 123 121 120 122 116 112 114 115 113 118 119 117 124 127 125 126
71 70 75 73 74 68 64 65 67 94 93 92 85 84 87 86 82 80 81 83 89 90 88
91 23 22 19 17 16 20 18 21 31 30 29 28 26 24 25 27 11 10 15 14
list 3: 0

2077.46
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 41 47 103 101 100 107 110 119 114 117 125 127 122 123 80 81 35
27 24 28 26 25 18 19 16 20 23 91 22 83 105 96 102 108 104 97 99 98 51
59 56 58 57 62 63 61 60 53 54 55 48 52 50 46 42 49 109 111 106 118 120
121 126 124 116 112 113 115 87 95 88 89 86 82 84 92 85 93 94 90 71 77
69 76 74 75 73 79 68 72 78 70 66 31 29 21 67 64 65 17 34 32 43 33 38
39 3 7 30 5 4 0 12 10 11 8 9 2 6 13 15 14 1 37 40 44 45 36
list 3: 0

1740.42
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 118 119 117 116 122 126 127 125 124 121 123 120 113 106 104
101 108 100 110 109 111 96 35 99 82 33 98 41 102 103 43 48 51 50 49 57
56 59 58 62 60 63 61 53 52 54 55 1 3 2 0 11 9 8 6 4 15 12 10 14 13 7 5
65 90 94 95 88 80 105 86 97 19 34 36 32 38 25 28 27 24 26 37 44 40 42
46 47 45 39 16 22 89 18 20 21 91 23 71 70 69 77 79 76 78 74 75 73 64
67 68 72 66 29 31 30 17 83 81 92 93 85 84 87 107 115 114 112
list 3: 0

AIM 8 PCB WORST TRADITIONAL

2298.02
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 86 85 82 83 81 89 88 80 92 84 91 90 94 38 37 45 47 2 4 6 7 5
15 13 11 14 10 12 0 8 3 1 9 61 56 58 49 51 50 54 55 52 48 60 62 57 59
19 25 27 26 31 29 30 24 28 17 18 16 20 22 21 23 63 53 95 93 33 41 44
46 42 43 36 39 40 34 32 35 73 75 74 78 72 77 76 79 69 71 65 64 70 66
68 111 110 100 103 102 101 109 108 96 98 99 105 104 97 107 106 67 127
117 116 119 118 112 114 115 113 123 120 121 125 124 122 126 87
list 3: 0

2522.67
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 74 76 72 68 77 71 69 70 66 67 65 64 73 75 78 79 31 30 27 26 25
17 19 12 10 9 8 11 16 28 24 29 21 23 22 20 18 14 4 0 2 1 3 6 7 15 13 5
```

```
46 45 47 39 36 32 41 42 44 40 35 33 43 37 38 34 58 62 63 59 57 51 50
54 48 49 56 60 52 55 53 61 106 107 104 98 123 102 96 99 105 97 113 118
117 119 115 114 112 116 122 120 121 124 126 125 127 103 100 110 109
108 111 101 91 90 88 94 87 93 80 84 92 95 85 86 82 89 81 83
list 3: 0

2662.36
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 28 17 25 24 18 16 10 14 2 3 6 5 7 4 12 8 1 0 9 11 22 19 21 20
23 13 15 46 44 47 40 41 43 33 35 32 45 42 36 37 39 108 110 96 97 99 98
100 111 102 127 118 117 119 125 115 121 120 123 122 113 112 114 116
124 126 103 109 101 104 106 105 107 38 34 59 58 62 56 57 49 60 61 53
52 48 50 51 54 55 63 95 84 85 93 92 80 75 73 72 69 71 70 67 64 66 76
68 65 74 79 77 78 82 83 81 91 89 94 90 88 86 87 27 26 31 30 29
list 3: 0

PAPM 2 PCB GLOBAL

10016.9
list 0: 1
list 1: 0 0
list 2: 2 24 14 12 31 4 30 28 21 7 20 22 11 27 0 26 9 1 17 16 29 5 6
15 13 23 10 8 25 3 19 18
list 3: 0 2 3 1

7079.23
list 0: 0
list 1: 180 180 list 2: 16 24 15 19 27 12 18 10 3 5 1 4 2 21 9 22 25
31 30 28 29 14 0 20 26 13 17 11 23 8 6 7
list 3: 1 0 2 3

4883.87
list 0: 0
list 1: 180 180
list 2: 27 26 25 10 19 30 24 11 21 14 20 29 9 22 12 5 1 23 16 28 31 8
17 13 18 15 3 4 2 6 0 7
list 3: 1 0 3 2

10640
list 0: 1
list 1: 0 0
list 2: 6 18 26 5 14 28 9 25 13 12 24 1 0 16 15 20 2 3 23 30 4 19 22
10 29 8 21 11 17 27 7 31
list 3: 3 0 1 2

PAPM 2 PCB BEST TRADITIONAL

9768.21
list 0: 1
list 1: 0 0
list 2: 17 3 2 10 24 13 6 14 28 5 4 30 8 25 0 11 26 27 16 15 12 31 23
7 21 22 29 20 9 19 1 18
list 3: 0 2 3 1

6931.34
list 0: 1
list 1: 0 0
list 2: 20 27 31 28 8 16 10 22 26 30 12 21 29 11 19 9 17 14 1 18 24 15
6 3 2 0 4 5 23 25 13 7
list 3: 0 1 2 3

4743.03
list 0: 0
```

```
list 1: 0 0
list 2: 3 22 15 0 1 4 17 13 23 30 27 26 24 25 9 5 18 31 10 6 21 14 16
12 20 28 11 7 19 29 8 2
list 3: 1 0 3 2

10710
list 0: 0
list 1: 0 0
list 2: 16 27 22 7 8 9 29 6 20 30 14 21 23 11 17 15 24 26 18 19 10 28
3 12 13 5 1 4 25 31 2
list 3: 1 2 3 0

PAPM 2 PCB WORST TRADITIONAL

10754
list 0: 0
list 1: 0 0
list 2: 15 8 0 11 9 1 18 20 14 24 13 16 31 28 21 22 6 2 10 27 12 26 30
29 5 25 23 4 3 17 7 19
list 3: 0 2 3 1

15296
list 0: 1
list 1: 0 0
list 2: 27 17 7 26 22 5 13 29 18 23 20 0 28 1 14 11 10 9 12 30 19 4 24
21 6 31 2 16 3 15 8 25
list 3: 2 3 0 1

19210.2
list 0: 0
list 1: 0 0
list 2: 3 14 18 23 7 26 4 27 28 21 30 6 25 15 0 24 13 8 10 2 9 1 31 12
17 20 29 22 5 16 11 19
list 3: 0 2 3 1

10749
list 0: 0
list 1: 0 0
list 2: 2 22 20 0 26 13 27 29 21 28 4 6 31 9 18 12 7 23 25 5 14 11 30
17 1 10 16 24 8 15 3 19
list 3: 0 3 2 1

PAPM 4 PCB GLOBAL

22377.8
list 0: 0
list 1: 0 270 270 270
list 2: 32 51 26 37 42 19 23 29 22 46 3 61 27 7 15 52 43 33 35 2 53 24
39 18 11 30 20 55 9 45 40 34 1 36 16 54 10 38 8 62 41 0 14 44 49 58 59
6 21 4 13 63 57 50 25 5 12 47 17 28 60 48 56 31
list 3: 3 1 2 0

16203
list 0: 0
list 1: 270 270 180 180
list 2: 1 52 18 60 0 38 10 54 48 33 58 3 4 5 6 7 55 32 43 24 21 45 41
42 57 49 17 14 37 8 22 44 11 30 29 62 19 31 53 2 20 28 47 40 9 12 39
56 50 34 25 61 35 26 15 23 13 36 27 46 59 51 16 63
list 3: 1 3 0 2

12580.8
list 0: 0
list 1: 270 270 180 180
list 2: 46 56 26 48 61 9 18 21 23 36 15 54 13 6 53 29 22 39 44 10 35
```

```
47 57 43 42 58 40 11 16 17 49 62 41 24 3 1 19 28 4 52 31 20 34 12 38
30 7 5 55 2 33 60 25 51 14 37 45 8 0 50 32 63 59 27
list 3: 1 0 3 2


22580.7
list 0: 0
list 1: 0 0 0 0
list 2: 63 34 10 21 57 6 24 33 59 27 0 29 52 14 25 41 19 60 39 47 23
45 55 22 8 17 36 30 44 31 15 9 49 46 35 62 3 12 5 13 1 40 2 4 61 43 38
26 32 18 48 16 28 20 53 11 37 54 58 7 56 42 50
list 3: 1 2 3 0

PAPM 4 PCB BEST TRADITIONAL

22078.4
list 0: 0
list 1: 0 0 0 0
list 2: 24 26 28 8 45 6 14 30 11 13 15 62 5 61 23 42 32 19 21 43 36 35
49 7 63 38 34 0 12 31 57 39 55 54 20 58 9 44 18 25 40 2 46 51 52 17 56
37 50 1 47 3 41 48 53 4 60 33 16 59 22 10 29 27
list 3: 3 2 1 0

16000.8
list 0: 0
list 1: 0 0 0 0
list 2: 0 5 23 2 6 20 3 4 38 47 41 8 54 12 55 61 60 30 18 33 56 46 9
21 1 7 53 62 10 36 58 43 27 16 34 13 39 44 24 17 14 22 19 29 50 15 37
59 26 51 32 57 40 42 25 49 63 11 52 35 45 28 48 31
list 3: 1 0 3 2

12393
list 0: 0
list 1: 0 0 0 0
list 2: 7 4 38 50 62 58 10 13 22 20 53 18 29 5 6 23 36 35 16 37 33 48
49 17 19 3 46 11 47 25 2 31 52 40 8 14 39 1 45 59 24 30 44 9 60 51 12
61 42 41 43 56 57 27 28 21 54 15 55 32 34 0 63 26
list 3: 1 3 0 2

22344.8
list 0: 0
list 1: 0 0 0 0
list 2: 34 6 8 33 43 11 49 62 14 21 45 23 53 27 13 12 36 39 54 55 0 5
29 24 56 3 9 20 37 22 48 35 31 32 63 38 30 28 52 58 51 47 18 1 4 40 19
59 42 26 25 57 10 44 7 17 41 2 61 15 60 46 50 16
list 3: 2 1 3 0

PAPM 4 PCB WORST TRADITIONAL

22121.7
list 0: 0
list 1: 0 0 0 0
list 2: 42 5 18 12 32 7 48 10 30 46 20 14 29 15 2 31 47 52 44 54 11 34
41 40 53 24 60 25 50 39 23 28 63 43 22 45 36 55 9 61 59 58 26 19 1 17
62 57 4 33 6 0 16 13 3 35 56 27 49 38 37 21 8 51
list 3: 0 3 2 1

28393.6
list 0: 0
list 1: 0 0 0 0
list 2: 34 53 6 59 46 23 16 35 54 13 58 44 38 22 0 40 4 1 27 49 63 15
11 56 2 24 17 3 42 47 20 48 60 12 50 14 9 10 8 41 21 31 43 55 29 33 52
45 7 25 26 18 61 30 32 36 37 39 5 57 62 28 19 51
list 3: 2 0 3 1
```

```
32595.7
list 0: 0
list 1: 0 0 0 0
list 2: 50 63 21 34 29 19 24 3 41 28 36 15 43 45 46 30 49 55 44 13 0
26 35 20 48 6 8 10 27 18 57 53 38 14 9 40 62 23 32 31 1 42 52 54 7 11
16 2 58 5 59 37 12 56 4 33 39 47 61 60 22 17 25 51
list 3: 2 0 1 3

22487.1
list 0: 0
list 1: 0 0 0 0
list 2: 48 42 6 9 17 57 40 2 44 31 4 61 62 46 63 54 15 25 14 13 37 7
33 50 27 28 49 23 52 60 22 12 29 21 20 56 3 24 41 47 35 58 43 38 18 32
34 36 51 59 39 10 53 30 0 8 5 45 55 11 1 19 26 16
list 3: 1 3 2 0

PAPM 8 PCB GLOBAL

45072.6
list 0: 33
list 1: 180 0 0 0 0 0 0 0
list 2: 76 90 121 95 1 91 40 92 98 110 116 84 25 33 32 19 34 12 89 112
111 21 61 52 118 28 64 43 66 3 8 120 63 103 23 126 109 54 6 57 4 42 97
79 104 70 80 58 86 0 75 105 93 83 27 113 20 124 96 85 24 31 30 60 38 2
56 117 5 26 35 49 37 46 67 123 15 48 71 18 108 100 101 78 99 47 29 45
125 13 106 68 122 36 17 11 59 102 87 51 55 7 114 65 107 127 14 10 41
62 39 115 69 50 53 119 22 44 94 16 81 73 72 9 88 77 74 82
list 3: 2 0 3 1

36527.2
list 0: 12
list 1: 180 180 180 180 180 180 180 180
list 2: 115 42 16 113 108 94 91 93 73 12 48 17 10 86 74 25 33 107 77
11 39 38 84 89 104 79 41 51 71 121 13 36 82 90 112 61 54 49 34 101 111
29 22 64 27 35 98 44 37 102 46 2 68 88 59 0 119 28 83 75 63 65 123 110
95 9 18 81 58 7 52 50 100 125 43 24 55 80 72 120 109 92 26 67 40 4 116
122 31 23 32 53 3 6 21 96 127 56 105 124 14 87 8 20 114 57 97 47 85 19
1 69 66 60 103 15 117 76 78 45 5 70 118 30 99 126 106 62
list 3: 1 0 2 3

31567
list 0: 8
list 1: 180 180 180 180 180 180 180 180
list 2: 79 41 52 37 116 123 24 60 97 8 49 86 80 44 38 69 13 113 91 106
89 126 75 88 125 107 43 84 32 22 51 36 5 68 61 2 28 96 121 10 98 56 20
7 117 90 73 124 104 58 82 31 100 122 46 71 62 115 57 21 0 14 87 48 101
9 78 127 45 118 110 40 102 42 53 23 16 77 47 70 63 85 92 109 26 35 119
59 114 12 34 54 103 25 18 83 81 65 30 112 76 93 120 105 72 94 15 55 4
50 3 66 95 19 64 108 11 17 1 33 6 99 74 111 27 67 29 39
list 3: 1 0 3 2

46384.3
list 0: 1
list 1: 180 180 180 180 180 180 0 0
list 2: 109 44 48 34 15 3 23 7 54 43 6 62 46 111 8 11 102 63 39 99 26
110 25 119 29 78 93 41 112 53 50 126 1 2 59 71 97 69 49 55 14 35 98
108 72 61 66 36 115 77 104 116 0 127 80 12 19 67 21 42 18 103 58 107
117 75 76 113 79 51 22 114 64 56 106 9 83 20 90 57 16 91 105 121 40 47
86 88 28 125 24 118 37 5 94 60 96 124 101 65 73 84 100 17 32 70 89 45
10 95 85 4 122 68 120 92 81 13 87 33 82 52 38 31 74 27 30 123
list 3: 0 3 1 2

PAPM 8 PCB BEST TRADITIONAL
```

```
44810.5
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 127 23 126 47 99 16 35 97 85 46 93 80 11 123 95 17 66 74 73 57
70 75 26 63 69 113 84 112 117 87 96 15 91 27 125 36 98 21 115 78 10
104 4 43 83 90 65 24 3 40 0 25 49 37 77 32 92 120 62 100 103 31 30 12
58 110 39 79 29 124 6 72 9 107 45 14 89 2 122 108 101 86 59 52 55 53
54 119 111 76 18 1 105 7 41 82 81 50 38 94 42 56 20 61 118 71 121 22
44 60 13 8 88 51 28 67 33 64 106 48 68 19 109 116 102 5 34 114
list 3: 2 0 3 1

36268
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 95 79 77 72 45 66 13 118 47 82 46 2 71 70 88 31 67 12 85 96 8
116 120 10 86 35 1 4 23 55 54 51 62 99 91 16 32 103 74 89 29 38 119
113 61 30 7 101 106 11 97 107 9 81 92 15 36 98 26 68 108 41 125 93 28
115 73 25 17 124 111 109 121 14 53 65 110 104 57 87 80 56 102 43 76
122 105 58 64 114 127 90 18 19 20 21 22 117 24 94 50 49 83 44 75 42 84
40 37 3 52 63 48 5 100 27 34 33 112 123 126 60 0 6 39 69 78 59
list 3: 1 0 3 2

31389.9
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 30 17 64 125 27 0 95 108 25 96 40 71 115 57 55 23 80 12 82 14
60 7 6 49 4 3 81 44 86 92 13 118 28 66 112 93 119 15 21 67 63 20 1 97
90 120 121 42 117 127 94 126 72 79 110 41 103 74 11 16 77 32 116 78
111 75 56 101 58 68 61 48 98 62 114 59 35 38 100 123 122 106 10 51 39
52 99 91 104 105 107 9 47 53 54 85 29 69 109 43 5 22 2 113 88 50 36 70
73 8 76 26 46 37 102 24 124 31 84 18 83 34 33 87 19 65 89 45
list 3: 1 0 3 2

46330.6
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 127 46 56 41 10 68 19 13 72 108 55 37 65 70 74 88 7 8 40 11 84
63 118 43 15 44 123 1 107 57 53 25 59 69 4 76 24 51 89 14 92 30 120 95
75 58 45 111 23 29 52 9 104 77 62 96 97 26 90 42 124 49 109 87 83 67
39 110 99 32 64 122 119 105 66 79 12 71 50 81 38 34 126 78 116 61 17 2
91 106 102 125 98 33 21 16 0 114 113 101 117 28 6 100 93 20 22 35 121
47 31 3 36 94 82 48 85 103 73 5 60 18 27 54 80 86 115 112
list 3: 2 0 3 1

PAPM 8 PCB WORST TRADITIONAL

45519.8
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 1 79 80 106 118 4 108 87 52 107 70 65 81 96 68 5 93 66 36 41
111 53 9 95 19 30 90 105 85 24 44 43 61 16 29 58 109 100 71 104 102 39
45 28 89 112 38 25 120 49 42 62 27 14 78 34 122 64 103 23 26 15 110 84
101 7 31 63 56 35 11 47 46 94 51 59 3 60 126 97 6 124 83 48 18 32 125
37 72 99 20 127 50 123 119 55 121 114 54 73 113 116 117 86 98 21 8 13
77 17 74 0 76 33 40 10 92 82 22 88 91 2 12 57 75 67 69 115
list 3: 3 2 0 1

55049.7
list 0: 24
list 1: 0 0 0 0 0 0 0 0
list 2: 34 78 49 109 70 65 96 55 0 10 60 28 9 27 11 40 93 81 19 58 125
66 21 43 41 88 124 36 47 92 5 90 64 48 62 61 108 103 102 32 95 67 85
118 22 26 63 127 20 73 114 80 99 39 45 75 82 4 72 53 14 94 7 59 91 30
```

```
74 17 120 33 76 77 1 57 18 104 6 25 89 83 51 13 106 117 69 16 8 29 15
56 50 79 100 37 44 126 84 111 116 71 68 23 110 87 113 3 12 122 2 24
107 86 97 52 112 115 119 38 123 98 101 54 46 42 31 105 35 121
list 3: 2 3 0 1

60997.4
list 0: 18
list 1: 0 0 0 0 0 0 0 0
list 2: 60 127 38 16 66 57 44 0 35 77 121 39 25 32 49 79 126 58 109
117 69 73 68 87 104 20 50 90 76 85 116 27 94 91 11 114 4 1 2 3 51 15
110 53 93 56 78 72 59 95 74 108 100 103 101 23 112 22 13 18 124 40 62
42 29 83 106 26 98 54 43 81 6 67 37 65 84 75 55 14 113 89 8 61 88 9 34
30 80 118 7 82 46 111 5 64 70 120 41 92 122 52 45 96 119 123 71 102 36
125 10 99 105 31 48 28 17 97 21 12 19 47 33 63 107 86 24 115
list 3: 0 3 2 1

46289.7
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 58 63 78 76 34 33 112 52 101 106 108 8 57 91 98 15 65 40 85 95
51 122 84 127 54 48 77 74 50 70 102 119 7 105 30 0 97 118 39 38 99 100
26 81 27 125 86 46 13 9 5 1 61 117 47 126 104 68 4 73 11 69 90 67 2
113 79 107 116 36 25 89 35 23 22 14 93 18 21 82 87 94 103 56 55 123 3
53 10 121 88 72 75 62 43 19 64 28 110 124 111 120 83 71 16 17 80 114
59 60 115 44 45 37 49 6 109 66 24 42 29 32 41 96 92 12 20 31
list 3: 1 3 0 2

RTHM 2 PCB GLOBAL

6.39428
list 0: 0
list 1: 180 0
list 2: 10 11 8 5 7 6 4 9 0 2 3 1 12 13 15 14 30 29 31 28 16 18 17 19
25 20 23 21 22 24 27 26
list 3: 3 0 2 1

6.39047
list 0: 0
list 1: 180 180
list 2: 4 6 12 15 5 7 0 2 3 1 8 9 10 11 19 16 24 25 27 26 17 18 23 21
20 31 29 30 28 22 14 13
list 3: 3 1 0 2

6.40565
list 0: 0
list 1: 0 0
list 2: 25 30 29 31 28 24 21 23 22 18 17 19 16 20 26 7 2 5 8 10 15 13
12 11 14 9 4 0 1 3 6 27
list 3: 0 1 2 3

6.90165
list 0:  0
list 1:  180 0
list 2:  11 15 14 13 12 16 17 18 19 23 22 21 20 24 25 26 27 31 30 29
28 0 1 2 3 7 6 5 4 8 9 10
list 3:  3 2 1 0

RTHM 2 PCB BEST TRADITIONAL

6.39047
list 0: 0
list 1: 0 0
list 2: 4 14 9 10 11 12 13 8 15 5 6 2 7 19 17 18 16 22 21 31 29 24 25
26 28 27 30 20 23 3 0 1
```

```
list 3: 2 3 1 0

6.39047
list 0: 0
list 1: 0 0
list 2: 30 25 26 27 28 24 18 19 20 23 16 17 22 21 29 31 7 4 14 9 11 8
13 15 10 12 5 0 3 1 2 6
list 3: 2 3 1 0

6.39428
list 0: 0
list 1: 0 0
list 2: 26 25 27 24 28 21 22 23 16 19 18 17 20 29 30 7 5 0 1 3 2 8 9
14 11 10 12 13 15 4 6 31
list 3: 0 1 3 2

6.93714
list 0:  0
list 1:  0 0
list 2:  30 29 28 24 20 16 17 18 19 23 22 21 25 26 27 7 11 10 15 14 13
12 8 9 6 5 4 0 1 2 3 31
list 3:  3 2 1 0

RTHM 2 PCB WORST TRADITIONAL

6.39047
list 0: 0
list 1: 0 0
list 2: 27 26 25 18 17 19 16 24 22 20 30 29 28 23 21 31 7 12 4 14 5 15
13 6 8 10 1 2 0 11 9 3
list 3: 0 2 1 3

8.12449
list 0: 0
list 1: 0 0
list 2: 13 12 8 2 0 7 6 5 4 3 1 10 11 9 14 31 28 25 27 26 16 22 23 21
20 18 17 19 24 29 30 15
list 3: 1 0 2 3

9.89017
list 0: 0
list 1: 0 0
list 2: 3 4 5 15 30 28 27 26 25 24 22 17 19 20 16 18 23 29 21 31 13 8
14 10 9 11 12 7 0 6 2 1
list 3: 2 3 1 0

7.18628
list 0: 0
list 1: 0 0
list 2: 11 15 14 13 12 16 17 18 19 23 22 21 20 25 24 26 27 31 30 29 28
4 0 1 2 3 7 6 5 8 9
list 3: 0 1 2 3
list 4: 0 0 1 0 0 0

RTHM 4 PCB GLOBAL

13.3825
list 0: 0
list 1: 270 90 180 0
list 2: 51 63 56 60 0 4 14 10 12 6 15 26 18 17 16 21 31 28 22 19 23 20
25 24 27 29 30 9 7 11 2 5 13 8 35 38 39 44 47 40 45 46 41 36 43 42 32
34 33 3 1 37 52 53 58 59 57 61 62 54 49 55 48 50
list 3: 3 2 1 0
```

```
10.5397
list 0: 0
list 1: 0 270 90 180
list 2: 9 14 11 0 1 4 5 6 7 2 33 36 34 39 37 32 44 45 47 46 42 40 43
41 35 38 49 50 53 52 55 48 58 61 63 62 56 59 60 57 51 54 17 22 23 18
21 20 26 30 29 24 31 28 25 27 16 19 3 8 13 15 12 10
list 3: 1 0 2 3

9.61975
list 0: 0
list 1: 0 270 90 180
list 2: 31 29 30 16 28 27 25 56 58 39 59 37 36 41 4 6 7 5 10 11 9 24
53 55 54 52 38 57 44 45 47 42 43 40 46 32 33 2 51 49 50 19 35 17 60 61
63 62 48 3 34 18 0 1 12 14 13 15 8 23 21 22 20 26
list 3: 1 2 3 0

14.2787
list 0:  0
list 1:  0 0 0 270
list 2:  36 37 35 34 33 32 63 62 58 59 55 54 51 50 49 48 52 53 56 57
61 60 19 18 17 16 20 21 22 23 27 26 25 24 28 29 30 31 3 2 1 0 4 5 6 7
11 10 9 8 12 13 14 15 40 44 45 46 47 43 41 42 39 38
list 3:  0 2 1 3

RTHM 4 PCB BEST TRADITIONAL

12.8516
list 0: 0
list 1: 0 0 0 0
list 2: 57 59 22 20 25 24 27 26 31 28 16 19 18 17 30 29 23 21 9 11 8
10 4 5 13 2 1 3 0 14 15 12 7 6 46 37 47 45 34 33 35 32 44 39 36 38 40
43 42 41 53 61 55 54 56 58 49 51 50 48 60 63 62 52
list 3: 2 1 3 0

11.0405
list 0: 0
list 1: 0 0 0 0
list 2: 36 39 32 33 43 40 42 47 46 45 44 41 34 49 51 57 50 54 53 52 55
48 56 62 60 63 61 58 59 2 9 10 15 12 11 14 13 8 1 4 6 5 7 0 3 22 23 21
20 17 16 29 26 31 30 24 27 28 25 18 19 35 38 37
list 3: 3 2 0 1

9.6
list 0: 0
list 1: 0 0 0 0
list 2: 15 10 23 8 5 0 3 6 42 47 46 45 44 43 13 12 41 40 4 7 2 19 32
48 33 50 51 34 37 38 39 57 36 35 17 52 49 54 56 61 62 63 58 59 60 27
55 16 18 1 20 21 22 24 29 30 31 26 53 25 28 11 9 14
list 3: 0 1 2 3

14.6434
list 0:  0
list 1:  0 0 0 0
list 2:  17 18 19 23 22 21 20 24 25 26 27 31 30 29 28 52 53 48 49 50
51 55 54 59 58 57 56 60 61 62 63 32 33 34 35 39 43 42 38 37 36 40 41
44 45 46 47 4 5 9 8 12 13 14 10 6 1 0 2 3 7 11 15 16
list 3:  3 0 1 2

RTHM 4 PCB WORST TRADITIONAL

12.9048
list 0: 0
list 1: 0 0 0 0
list 2: 27 28 29 31 63 62 61 60 56 59 57 58 52 55 48 54 49 51 50 53 41
```

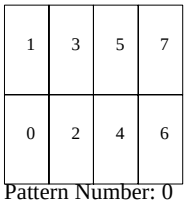
```
43 46 47 45 44 42 36 35 32 39 38 40 37 34 33 3 2 5 7 0 1 6 8 4 10 14
13 15 12 11 9 30 24 21 18 17 19 22 16 23 20 25 26
list 3: 0 1 2 3

15.1973
list 0: 0
list 1: 0 0 0 0
list 2: 29 58 59 57 56 62 52 60 55 61 53 50 48 49 51 54 63 47 42 41 43
40 44 39 34 32 35 33 36 38 37 46 45 11 9 8 13 6 12 1 3 0 2 7 4 15 10
14 5 16 23 21 22 17 18 19 20 31 26 25 24 27 28 30
list 3: 2 3 1 0

16.6662
list 0: 0
list 1: 0 0 0 0
list 2: 22 23 18 17 19 16 30 31 24 25 20 21 5 4 7 6 8 10 15 12 2 1 3 0
13 14 9 11 37 39 36 38 40 47 46 44 33 34 35 32 45 42 41 43 55 54 53 52
58 60 50 51 49 48 61 63 56 57 59 62 29 28 27 26
list 3: 0 3 2 1

14.3621
list 0:  0
list 1:  0 0 0 0
list 2:  23 19 3 2 1 0 4 5 6 7 10 9 8 12 13 14 15 11 32 33 35 34 39 38
36 37 42 41 40 44 45 46 47 43 48 49 51 50 55 54 53 52 58 57 56 60 61
62 63 59 16 17 18 21 20 25 24 28 29 30 31 27 26 22
list 3:  3 0 1 2

RTHM 8 PCB GLOBAL

24.9589
list 0: 12
list 1: 180 0 180 0 180 0 0 0
list 2: 0 7 6 4 1 12 15 27 29 31 24 45 47 62 49 60 63 58 59 57 56 61
48 50 53 55 54 52 51 39 38 70 66 71 69 77 79 96 98 100 103 101 102 97
108 105 107 104 106 74 73 76 65 68 67 64 78 72 75 111 109 110 99 117
118 124 115 112 116 122 121 123 120 127 113 119 114 125 126 89 90 91
88 95 92 81 82 83 84 85 86 87 80 93 94 43 46 42 41 40 44 33 37 34 32
36 35 23 20 16 18 21 22 19 17 28 25 26 30 10 8 11 9 14 13 5 2 3
list 3: 2 3 0 1

26.5157
list 0: 8
list 1: 0 180 0 0 0 180 180 180
list 2: 49 48 54 53 52 55 71 65 73 43 42 15 14 6 3 26 31 20 21 18 23
22 17 19 16 29 28 27 25 30 24 11 8 12 13 0 46 32 35 33 38 39 40 45 34
44 47 41 9 10 1 7 4 5 2 37 36 87 85 64 68 75 72 76 66 82 70 84 80 94
95 67 69 86 81 92 91 112 126 120 122 123 121 113 124 114 119 115 79 74
77 83 127 116 105 106 109 108 102 100 107 110 111 104 125 117 118 103
101 98 96 97 99 90 93 88 89 78 51 57 58 60 56 59 62 63 61 50
list 3: 1 0 3 2

24.1735
list 0: 22
list 1: 0 0 180 180 0 180 180 0
list 2: 50 55 48 51 52 123 112 113 115 116 121 125 59 56 58 62 60 46
27 30 31 12 14 0 1 3 4 2 7 5 8 9 10 13 15 11 6 25 47 45 41 38 19 92 73
68 70 71 64 74 77 91 69 81 67 87 82 83 66 65 80 85 72 78 75 76 104 111
103 96 102 99 97 98 101 100 107 106 109 110 108 105 79 89 88 84 90 94
86 20 35 16 34 39 17 18 33 32 37 24 21 22 95 93 29 28 26 23 44 42 40
36 43 53 61 63 126 127 120 114 117 119 118 122 124 57 54 49
list 3: 0 1 2 3

29.8496
```

```
list 0: 0
list 1: 180 0 0 0 180 0 180 180
list 2: 66 65 64 99 97 103 69 74 75 39 38 37 40 41 43 71 70 73 72 107
68 102 98 96 101 100 104 105 106 110 111 76 77 78 79 93 94 91 95 114
108 109 112 116 113 115 119 118 117 120 121 122 123 87 86 85 80 81 83
127 126 124 125 82 51 50 84 55 54 53 52 19 48 49 18 17 16 21 23 22 20
26 25 29 30 13 12 31 60 61 32 33 34 35 62 63 92 59 90 89 88 58 57 56
27 24 28 15 14 11 3 0 1 2 7 8 36 9 10 6 5 4 44 45 42 46 47 67
list 3: 2 0 1 3

RTHM 8 PCB BEST TRADITIONAL

25.5978
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 50 54 55 61 107 108 103 100 97 99 104 110 101 98 96 102 109
111 106 105 124 119 114 113 115 112 126 116 118 117 127 125 122 120
121 123 79 75 73 72 74 78 69 77 76 71 70 64 66 68 94 90 91 92 95 85 86
80 81 83 84 82 87 93 88 89 25 28 23 29 21 30 27 26 24 31 20 19 17 22
16 18 67 65 53 51 48 34 33 32 42 41 40 36 37 44 39 62 63 49 1 0 5 8 10
9 2 3 6 4 14 11 12 15 13 7 35 38 45 47 46 43 59 57 56 58 60 52
list 3: 2 3 1 0

25.3441
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 18 35 20 30 39 3 4 12 8 11 14 0 1 2 6 5 7 13 15 10 9 45 42 40
47 46 56 61 63 53 55 51 49 50 32 83 82 87 93 88 16 36 37 17 22 34 33
81 80 96 103 54 48 52 62 58 59 60 57 43 41 25 89 90 73 78 77 68 67 65
64 71 70 66 69 79 74 75 72 76 94 95 92 107 105 104 106 109 110 118 115
114 113 112 116 127 126 124 121 120 123 122 125 117 119 111 108 102 19
84 85 91 29 21 23 28 26 31 24 27 44 38 98 101 100 86 99 97
list 3: 0 1 3 2

22.4523
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 18 35 20 30 39 3 4 12 8 11 14 0 1 2 6 5 7 13 15 10 9 45 42 40
47 46 56 61 63 53 55 51 49 50 32 83 82 87 93 88 16 36 37 17 22 34 33
81 80 96 103 54 48 52 62 58 59 60 57 43 41 25 89 90 73 78 77 68 67 65
64 71 70 66 69 79 74 75 72 76 94 95 92 107 105 104 106 109 110 118 115
114 113 112 116 127 126 124 121 120 123 122 125 117 119 111 108 102 19
84 85 91 29 21 23 28 26 31 24 27 44 38 98 101 100 86 99 97
list 3: 0 1 3 2

29.9074
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 52 53 49 50 20 22 4 23 0 1 19 18 68 72 73 70 71 66 100 104 108
83 82 81 80 67 65 69 74 75 88 84 85 86 87 91 90 89 95 94 93 92 79 78
77 76 2 3 7 6 5 10 8 27 26 28 34 35 29 30 31 12 13 14 15 11 9 25 32 51
54 55 36 59 44 63 62 61 60 115 112 99 102 103 116 117 119 123 122 121
118 114 56 57 58 40 41 42 46 45 47 43 39 38 37 33 24 21 17 16 64 96 48
97 98 101 106 107 111 110 109 105 120 124 125 126 127 113
list 3: 2 0 1 3

RTHM 8 PCB WORST TRADITIONAL

28.2402
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 106 97 107 105 104 100 101 102 110 108 111 109 103 123 121 115
113 112 114 119 124 117 120 116 127 126 125 118 122 50 57 42 43 41 32
37 39 44 36 46 60 61 55 23 29 20 28 14 13 8 3 75 67 70 77 78 79 76 71
```

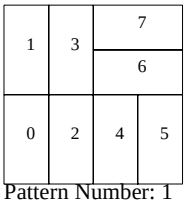
```
69 68 74 73 66 65 72 84 85 86 88 91 81 82 83 96 98 99 56 54 53 52 62
63 58 48 51 49 59 33 34 35 40 38 45 47 31 27 25 19 17 26 30 15 7 10 11
4 12 5 6 0 2 1 9 21 22 24 18 16 64 90 87 92 95 93 94 80 89
list 3: 0 2 1 3

30.0878
list 0:  0
list 1:  0 0 0 0 0 0 0 0
list 2:  120 123 119 116 115 124 114 112 121 122 117 126 125 94 81 76
93 82 86 88 91 90 92 95 80 83 79 66 67 68 64 71 77 78 69 72 74 75 73
70 65 87 84 89 85 24 23 17 16 22 25 30 18 15 14 1 21 26 27 29 28 31 20
19 12 13 0 7 2 3 9 6 5 11 10 8 4 43 33 44 32 36 58 55 54 48 51 50 52
59 57 53 62 60 63 61 49 56 38 40 41 42 47 37 46 45 39 35 34 96 99 102
104 105 106 107 101 97 108 109 111 110 100 103 98 127 113 118
list 3:  3 0 1 2

29.631
list 0:  0
list 1:  0 0 0 0 0 0 0 0
list 2:  52 61 63 59 58 56 34 38 39 47 94 92 85 86 79 76 64 68 67 70
72 73 75 78 77 74 71 69 66 65 87 91 83 84 89 95 90 80 82 81 88 93 31
24 22 18 16 19 17 20 21 23 26 27 25 15 10 5 7 2 3 4 9 14 13 8 11 28 30
29 12 0 1 6 42 41 33 35 36 32 37 43 45 46 40 44 62 53 54 57 60 106 96
103 112 117 114 118 120 127 124 125 122 116 123 126 121 119 115 113 98
99 97 102 107 110 104 111 108 109 101 100 105 55 50 49 48 51
list 3:  0 1 2 3

30.1979
list 0: 0
list 1: 0 0 0 0 0 0 0 0
list 2: 93 92 88 84 86 83 64 65 66 67 12 13 14 15 28 30 26 27 25 11 10
9 8 6 4 5 0 1 2 3 16 17 21 18 45 46 47 56 57 62 61 23 19 44 41 40 36
37 32 33 34 38 42 43 52 39 35 48 49 50 53 54 58 60 22 20 7 24 29 31 63
79 78 75 74 91 90 89 85 80 81 82 87 68 69 70 71 73 72 95 107 106 105
104 123 119 100 96 97 102 103 94 76 77 111 110 109 59 55 125 124 51
126 127 108 122 121 120 116 118 115 114 113 112 117 101 98 99
list 3: 3 1 0 2
```

Appendix F

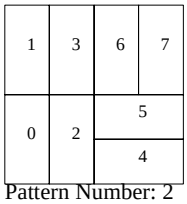
Pattern alternatives for 8 identical PCB and PAPM.



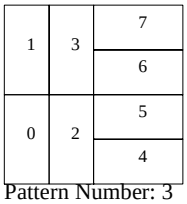
Pattern Number: 0
pcbOO: {0, 0, 0, 0, 0, 0, 0, 0}



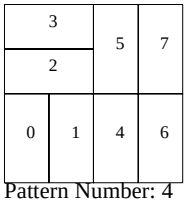
Pattern Number: 1
pcbOO: {0, 0, 0, 0, 0, 0, 1, 1}



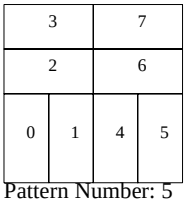
Pattern Number: 2
pcbOO: {0, 0, 0, 0, 1, 1, 0, 0}



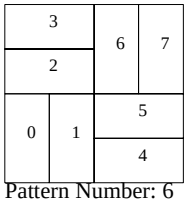
Pattern Number: 3
pcbOO: {0, 0, 0, 0, 1, 1, 1, 1}



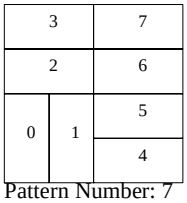
Pattern Number: 4
pcbOO: {0, 0, 1, 1, 0, 0, 0, 0}



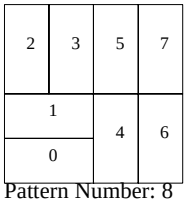
Pattern Number: 5
pcbOO: {0, 0, 1, 1, 0, 0, 1, 1}



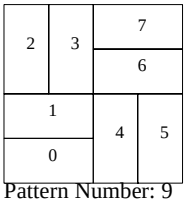
Pattern Number: 6
pcbOO: {0, 0, 1, 1, 1, 1, 0, 0}



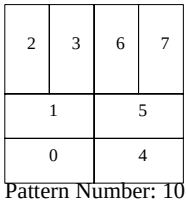
Pattern Number: 7
pcbOO: {0, 0, 1, 1, 1, 1, 1, 1}



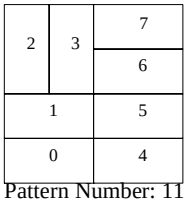
Pattern Number: 8
pcbOO: {1, 1, 0, 0, 0, 0, 0, 0}



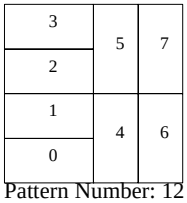
Pattern Number: 9
pcbOO: {1, 1, 0, 0, 0, 0, 1, 1}



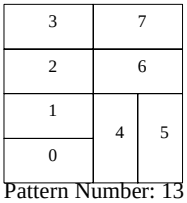
Pattern Number: 10
pcbOO: {1, 1, 0, 0, 1, 1, 0, 0}



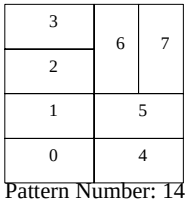
Pattern Number: 11
pcbOO: {1, 1, 0, 0, 1, 1, 1, 1}



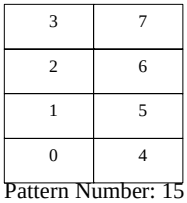
Pattern Number: 12
pcbOO: {1, 1, 1, 1, 0, 0, 0, 0}



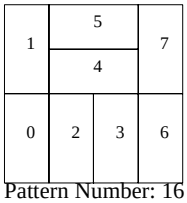
Pattern Number: 13
pcbOO: {1, 1, 1, 1, 0, 0, 1, 1}



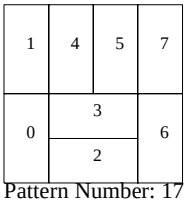
Pattern Number: 14
pcbOO: {1, 1, 1, 1, 1, 1, 0, 0}



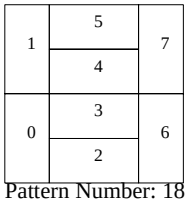
Pattern Number: 15
pcbOO: {1, 1, 1, 1, 1, 1, 1, 1}



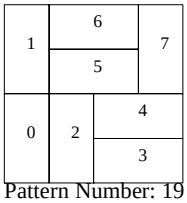
Pattern Number: 16
pcbOO: {0, 0, 0, 0, 1, 1, 0, 0}



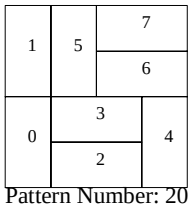
Pattern Number: 17
pcbOO: {0, 0, 1, 1, 0, 0, 0, 0}



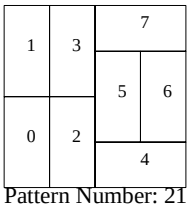
Pattern Number: 18
pcbOO: {0, 0, 1, 1, 1, 1, 0, 0}



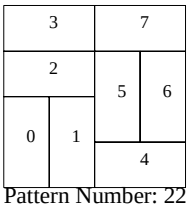
Pattern Number: 19
pcbOO: {0, 0, 0, 1, 1, 1, 1, 0}



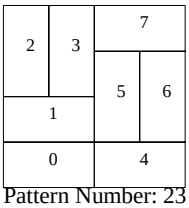
Pattern Number: 20
pcbOO: {0, 0, 1, 1, 0, 0, 1, 1}



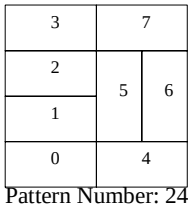
Pattern Number: 21
pcbOO: {0, 0, 0, 0, 1, 0, 0, 1}



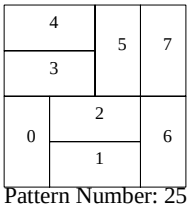
Pattern Number: 22
pcbOO: {0, 0, 1, 1, 1, 0, 0, 1}



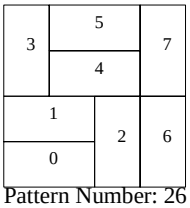
Pattern Number: 23
pcbOO: {1, 1, 0, 0, 1, 0, 0, 1}



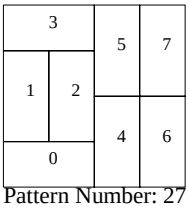
Pattern Number: 24
pcbOO: {1, 1, 1, 1, 1, 0, 0, 1}



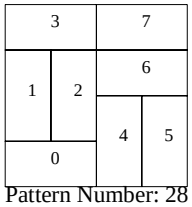
Pattern Number: 25
pcbOO: {0, 1, 1, 1, 1, 0, 0, 0}



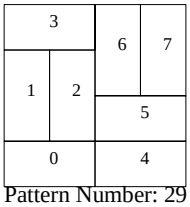
Pattern Number: 26
pcbOO: {1, 1, 0, 0, 1, 1, 0, 0}



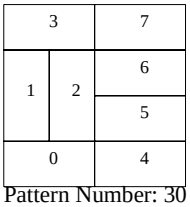
Pattern Number: 27
pcbOO: {1, 0, 0, 1, 0, 0, 0, 0}



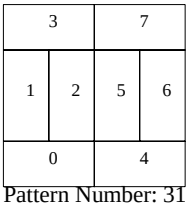
Pattern Number: 28
pcbOO: {1, 0, 0, 1, 0, 0, 1, 1}



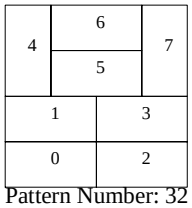
Pattern Number: 29
pcbOO: {1, 0, 0, 1, 1, 1, 0, 0}



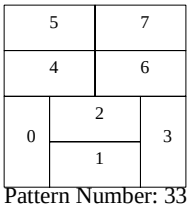
Pattern Number: 30
pcbOO: {1, 0, 0, 1, 1, 1, 1, 1}



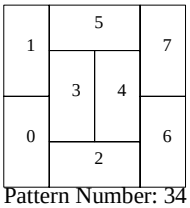
Pattern Number: 31
pcbOO: {1, 0, 0, 1, 1, 0, 0, 1}



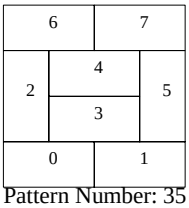
Pattern Number: 32
pcbOO: {1, 1, 1, 1, 0, 1, 1, 0}



Pattern Number: 33
pcbOO: {0, 1, 1, 0, 1, 1, 1, 1}



Pattern Number: 34
pcbOO: {0, 0, 1, 0, 0, 1, 0, 0}



Pattern Number: 35
pcbOO: {1, 1, 0, 1, 1, 0, 1, 1}

Appendix G

Solution Data for Experiment 5

The tables below show the Traditional Best, Traditional Worst, and global results for the respective analyses. The data is presented here to alleviate the mass of tables which would be otherwise required in Chapter 5.

Table G.1: Experiment 5 for AIM and 2 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
114	162.00	139.48	33.47	29.01	32.74
120	156.93	146.92	34.33	30.85	33.44
140	160.09	143.51	34.22	31.38	33.95
144	169.20	138.08	35.50	29.86	34.53
100	166.30	163.73	27.43	25.92	28.01
122	165.50	135.36	32.42	29.90	30.37
122	166.75	133.41	29.08	29.00	28.31
138	162.48	139.09	30.62	29.98	27.64
122	162.74	145.58	29.33	29.66	28.97
112	161.30	150.00	31.00	27.22	28.79

Table G.2: Experiment 5 for AIM and 4 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
108	472.10	376.48	34.12	28.29	29.73
116	474.71	373.99	34.15	30.59	32.66
128	462.82	385.78	41.01	29.35	31.00
132	482.48	366.22	34.32	28.51	30.08
100	467.49	383.49	26.64	27.00	25.98
116	438.72	410.22	34.93	30.22	28.71
116	432.77	415.83	36.39	29.73	31.40
128	448.79	400.90	33.76	30.36	30.75
116	477.37	372.55	34.07	29.54	30.12
108	483.74	370.77	32.16	27.53	29.82

Table G.3: Experiment 5 for AIM and 8 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	983.34	775.79	33.39	31.57	32.67
120	1009.73	743.65	38.75	27.72	29.82
144	790.37	785.22	35.58	30.18	29.82
144	917.75	799.34	34.18	28.19	31.23
96	974.03	779.37	35.68	28.64	34.78
112	898.91	830.20	33.85	31.32	30.06
112	967.33	779.81	31.79	29.99	27.16
128	983.33	752.58	33.07	26.51	26.81
112	1009.74	750.61	38.14	32.40	32.36
104	959.97	795.95	36.82	30.80	32.05

Table G.4: Experiment 5 for PAPM and 2 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	213.37	200.65	427.17	415.52	409.21
150	213.96	195.46	428.52	413.34	410.49
134	214.21	198.16	424.66	411.80	411.71
114	214.21	198.16	419.51	381.79	380.63
126	211.91	199.20	428.18	420.38	419.04
138	212.60	200.13	422.55	420.79	416.79
112	210.07	193.07	414.61	400.32	398.53
142	209.63	196.51	414.78	407.53	406.48
126	220.04	188.87	438.43	393.38	393.02
128	212.12	188.04	428.04	401.29	399.93

Table G.5: Experiment 5 for PAPM and 4 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	216.05	206.21	429.90	407.47	405.68
148	219.02	206.56	431.51	408.20	407.63
132	217.02	205.78	422.98	403.14	401.80
112	211.38	205.98	410.03	403.50	402.34
124	217.06	206.34	435.37	413.50	411.47
136	214.53	207.12	423.96	408.30	406.22
112	216.04	202.64	426.23	401.00	399.78
140	215.50	204.33	423.05	399.04	399.03
124	224.70	200.77	442.31	421.03	395.72
128	230.15	185.95	458.36	418.72	376.71

Table G.6: Experiment 5 for PAPM and 8 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	208.41	190.72	407.70	375.93	374.29
120	205.32	193.65	507.93	482.36	480.24
144	209.18	193.55	381.06	352.84	354.86
144	211.55	191.79	318.14	296.71	286.79
96	203.44	194.70	532.99	519.70	518.38
136	220.85	206.06	431.46	404.94	401.70
104	217.59	206.79	421.95	403.45	401.19
136	213.14	199.29	415.43	391.49	389.99
120	222.45	199.76	430.31	390.23	388.60
120	218.74	195.96	431.04	388.12	386.13

Table G.7: Experiment 5 for RTHM and 2 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	87.92	81.41	0.28	0.29	0.28
150	82.32	73.00	0.37	0.32	0.33
134	82.47	73.03	0.39	0.40	0.39
114	93.97	71.42	0.38	0.36	0.40
126	77.32	74.01	0.27	0.26	0.26
138	79.28	78.80	0.31	0.31	0.34
112	80.82	74.36	0.32	0.33	0.32
142	83.55	79.33	0.39	0.34	0.33
126	89.42	72.39	0.30	0.34	0.35
128	90.03	76.06	0.28	0.28	0.27

Table G.8: Experiment 5 for RTHM and 4 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	118.48	103.85	0.29	0.29	0.28
148	115.13	101.53	0.34	0.32	0.34
132	113.95	104.08	0.32	0.36	0.32
112	110.87	107.78	0.35	0.39	0.39
124	112.90	101.90	0.27	0.27	0.28
136	111.49	101.81	0.34	0.30	0.31
112	112.38	101.90	0.29	0.32	0.30
140	114.20	104.37	0.39	0.39	0.35
124	115.87	98.83	0.31	0.30	0.33
128	128.49	86.05	0.28	0.25	0.24

Table G.9: Experiment 5 for RTHM and 8 PCB.

No. of Comp	Estimated Worst	Estimated Best	Traditional Worst	Traditional Best	Global
112	122.13	105.20	0.28	0.26	0.25
152	117.04	104.81	0.45	0.39	0.41
136	119.87	103.77	0.34	0.35	0.34
112	121.56	103.41	0.44	0.39	0.38
128	112.94	103.16	0.30	0.27	0.28
136	116.75	103.18	0.33	0.32	0.29
112	115.78	103.06	0.31	0.27	0.29
144	118.06	103.94	0.35	0.33	0.34
128	120.70	104.14	0.29	0.29	0.30
112	117.46	104.64	0.33	0.29	0.30

Vita

John Tester earned his Doctorate of Philosophy for Industrial and Systems Engineering in 1999 from Virginia Polytechnic Institute and State University. He received his Masters of Science in Systems Engineering from the Air Force Institute of Technology and his Bachelors of Science for Mechanical Engineering from Tennessee Technological University.

He was an officer in the United States Air Force, serving as a materials, processes, mechanical, and systems engineer at Hill, Wright-Patterson, and Kirtland Air Force Bases. He was honorably discharged in 1991 with the rank of Captain. After his military service, John Tester worked in the commercial sector for 3 years at a mobile telecommunications firm, Amtech Systems Corporation. While at Amtech, he designed new commercial products; he also designed and installed automated manufacturing tooling and test equipment.