TETRIS-PACKING PROBLEM WITH MAXIMIZING FILLED GRID SQUARES

A Thesis
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
In
Computer Science

By
Paul-John Alonte To
2006
SIGNATURE PAGE

THESIS: TETRIS-PACKING PROBLEM WITH MAXIMIZING FILLED GRID SQUARES

AUTHOR: Paul-John Alonte To

DATE SUBMITTED: ________________________________

Department of Computer Science

Dr. Gilbert Young
Thesis Committee Chair
Computer Science

Dr. Sang-Eon Park
Computer Science

Dr. Daisy Sang
Computer Science
ACKNOWLEDGMENTS

My thesis committee has supported my effort to write and publish this thesis on an aggressive timeline. A great deal of thanks goes out to Dr. Gilbert Young for his leadership, hard work, and especially for helping me realize what research is all about. I also thank Dr. Sang-Eon Park and Dr. Daisy Sang for their feedback and willingness to review my work while I was out of state.

Dr. David Liben-Nowell at Carleton College answered questions related to the major paper on the Tetris Problem, which he co-authored.

I also want to acknowledge the help of several fellow students at Cal Poly Pomona: Anthony Tam, for his Tetris wisdom and feedback on the effectiveness of the proposed heuristics; and Kevin Johnson, for suggesting resources on Java Swing.

I am also grateful to the following individuals that tested my software on their personal computers and provided invaluable feedback: Hemant Bhatia, Norberto Gaspar, Helen Ma, Jennifer Maldonado, and Anthony Tam. I am proud of having worked with these colleagues and friends.

Finally, I want to acknowledge the moral support given to me by my family and relatives—both here and in the Philippines. For that I am eternally grateful.
Tetris is a classic video game invented by Alexey Pazhitnov, a Russian mathematician, in the 1980’s. A player starts out with an empty vertical game board divided into unit squares. The game pieces consist of an endless random sequence of tetrominoes, or shapes made up of four unit squares arranged in various ways. As each piece falls, players must slide them left or right or rotate them, in order to form complete rows of unit blocks. Completed rows of unit grid squares are eliminated from the stack and a new empty row is created on top. The game ends when the height of the block stack prevents placing new pieces. Scoring is based on the number of rows eliminated. Breukelaar, et al. [3] recently formalized the game into the Tetris Problem, which they proved NP-hard. Their version allowed for game boards of arbitrary initial configuration, width, and height.

We present a new class of packing problems called the Tetris-Packing Problem (TPP). A problem instance is a 5-tuple $G = (\beta, (w, h), (P_1, P_2, \ldots, P_p), \{S_1, S_2, \ldots, S_s\}, u)$ where $\beta$ is the initial configuration of the game board (empty or arbitrary), $(w, h)$ specifies the dimensions of the game board, $(P_1, P_2, \ldots, P_p)$ is a sequence of $p$ game pieces each belonging to the shape set $\{S_1, S_2, \ldots, S_s\}$, and $u$ specifies if the top of the board is open or closed (whether pieces can be extended partially beyond its top). Unlike the game of Tetris, TPP has no line elimination. We prove that TPP is generally NP-hard with the objective function of number of filled grid squares. We describe T-PACK, a Java API for implementing and testing Tetris-Packing heuristics, with which we developed the Corner Fit, Deepest Fit, Hybrid Corner/Deepest Fit, and Modified Best Fit heuristics.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>Table of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 History of Bin Packing</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research Goals</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2 Literature Survey</td>
<td>4</td>
</tr>
<tr>
<td>2.1 The Classical Bin Packing</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Open-end Bin Packing</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Bin Packing in Higher Dimensions</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Best Fit, Next Fit, Random Fit, and Worst Fit</td>
<td>8</td>
</tr>
<tr>
<td>2.5 Refined heuristics</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Sum-of-Squares in Closed Bin-Packing</td>
<td>10</td>
</tr>
<tr>
<td>2.7 The game of Tetris</td>
<td>11</td>
</tr>
<tr>
<td>2.8 From the game of Tetris to the TETRIS Problem</td>
<td>12</td>
</tr>
<tr>
<td>2.9 Applet Environments</td>
<td>14</td>
</tr>
<tr>
<td>2.10 Open Software Standards</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 3 Defining Tetris-Packing</td>
<td>16</td>
</tr>
<tr>
<td>3.1 TETRIS-PACKING PROBLEM Terminology</td>
<td>16</td>
</tr>
</tbody>
</table>
Chapter 8 Conclusion

8.1 General remarks about the study

8.2 For further investigation

8.3 Other implementation challenges

8.4 An invitation

References

Appendix A Experiment 1 Data table (Rectangles)

Appendix B T-PACK manual
TABLE OF FIGURES

Figure 2-1. Optimal bin packing [9] ................................................................. 5
Figure 2-2. Two-dimensional bin packing .......................................................... 6
Figure 2-3. Two-dimensional open-end bin packing configurations .................. 7
Figure 2-4. Three-dimensional open-end bin packing configurations ................. 8
Figure 2-5. Tetrads ....................................................................................... 12
Figure 2-6. Falling Tetris piece is about to eliminate the bottom two lines ....... 13
Figure 4-1. Optimal solution to Closed TPP, $h=1$ ........................................... 23
Figure 4-2. Algorithm for Closed TPP with height 1 ......................................... 24
Figure 4-3. Reduction from PARTITION to Closed TPP, $h = 2$ ....................... 26
Figure 4-4. A "yes" instance of PARTITION ..................................................... 26
Figure 4-5. A "yes" instance of TPP .................................................................. 27
Figure 4-6. Extension of Theorem 2 for fixed $h \geq 2$ ......................................... 28
Figure 4-7. Reduction from PARTITION to Open-end TPP, $h=1$, rectangles .... 29
Figure 4-8. Another "yes" instance of PARTITION ............................................ 30
Figure 4-9. Another "yes" instance of TPP ......................................................... 30
Figure 5-1. Corner Fit .................................................................................... 34
Figure 5-2. GET_FREE_SPACES_FOR algorithm ............................................. 35
Figure 5-3. Operation of Corner Fit ................................................................. 36
Figure 5-4. Deepest Fit .................................................................................... 37
Figure 5-5. Operation of Deepest Fit ............................................................... 38
Figure 5-6. Hybrid Corner/Deepest Fit ............................................................ 39
Figure 5-7. Modified Best Fit ....................................................................... 40
CHAPTER 1 INTRODUCTION

Given a finite set of fixed-size objects and a finite set of bins of fixed capacity, the Bin Packing Problem (BPP) lies in finding an assignment of objects to bins, subject to the constraint that the total size of the items in a bin does not exceed the bin’s capacity, with the goal of optimizing some given objective function such as the number of bins used. At first glance, it may seem like a trivial exercise in packing containers, but bin packing actually has many practical applications. It bears striking similarities to problems of scheduling, semiconductor manufacturing, and resource management—all of which continue to allow today’s digital world to function.

The aim of this chapter is to introduce the reader to bin-packing problems and some relevant issues surrounding them. We begin by briefly describing its organization. In the first section, we will present the relevance and history of bin packing. The second section contains the motivation behind this thesis, and the third section states the research goals.

1.1 History of Bin Packing

In the 1970's, the world began to shift its eyes toward optimizing the division of resources and labor [3]. Reforms in the transportation, business operations, and technology industries were imminent. In the realm of business operations management, people responsible for the downsizing and reorganization of companies did not use automated methods to make business decisions, such as what resources were to be cut or what units were being underutilized. Rather, they relied solely on subjective, error-prone human factors. Around the same time, lower costs for computer hardware presented
cheaper resources for more expensive projects. As the cost of microprocessors continue to dwindle, a greater share of multi-processing applications emerge with the hope of beating important NP-hard problems such as the problem of scheduling.

1.2 Motivation

Bin packing problems are interesting problems in resource management, because they relate directly to problems of scheduling. Their commonalities include: (1) a finite amount of resources capable of doing work; (2) an indeterminable number of jobs or tasks that must be completed; and (3) the restrictions (availability, deadlines, size constraints, etc.) by which resources and tasks can be matched. The general problem deals with how to optimally assign tasks to resources, just as bin packing deals with packing a set of items (tasks) into bins (resources) which have a fixed size (constraints).

One can reinterpret a solution to bin packing as a solution to scheduling. The literature classifies scheduling problems as off-line or on-line. Off-line algorithms know about all the inputs, or jobs, to work on before starting. They can always make the “smartest” decisions in assigning tasks to available resources. Nevertheless, off-line scheduling is still NP-hard [11] in general. On-line algorithms, on the other hand, only know about the resources and types of tasks. Not knowing anything about the next input until it arrives, they must always make spot decisions that can have unforeseen negative effects. By virtue of uncertainty, on-line scheduling is at least NP-hard [11]. Although just as hard as their off-line counterparts, on-line problems relate to contemporary demands for important timely information. Scheduling is somewhat more tedious to deal with on a purely theoretical level, and therefore bin packing serves as a more intuitive staging area without loss of generality.
1.3 Research Goals

We propose and analyze a new class of packing problems and heuristics. Another goal is creating a Java software package for implementing and evaluating packing heuristics. By varying certain parameters, we hope to understand their effects on the problem’s complexity. Moreover, a Web interface will be available to allow the public to learn about and view these heuristics in action.
CHAPTER 2 LITERATURE SURVEY

This chapter is composed of a survey of the literature related to bin packing. The sources were taken from many sources, including published academic journals, conference proceedings, online journals, and various textbooks on programming.

The chapter is divided into two main areas. Sections 2.1-2.8 cover the evolution of the classical bin-packing problem to more recent packing problems. Sections 2.9-2.10 lay out the background for designing and engineering the software simulator to be used in evaluating the heuristics proposed in this thesis. Section 2.9 identifies key technologies that will be integrated into the final product. Section 2.10 goes into detail on designing software to be as an open standard by other programmers.

2.1 The Classical Bin Packing Problem

The Bin Packing Problem (BPP) is a problem in combinatorial optimization. Horowitz, et al. formally define BPP as follows,

We are given \( n \) objects that have to be placed in bins of equal capacity, \( L \).

Object \( i \) requires \( l_i \) units of bin capacity. The objective is to determine the minimum number of bins needed to accommodate all \( n \) objects. No object may be placed partly in one bin and partly in another. [6]

Combinatorial optimization can be interpreted in many ways and can model many situations involving selecting, reordering, or grouping objects to optimize (that is, maximize or minimize) some value functions [6]. The optimization version of the Bin-Packing Problem has been studied extensively as early as the 1960's. Since BPP was proven to be NP-hard, heuristics have been developed and applied toward it and other related problems. These problems encompass operational management, process
scheduling, and combinatorial optimization. For example, the instance of BPP in Figure 2-1 consists of placing six objects \( \{l_1, l_2, \ldots, l_n\} = \{5, 6, 3, 7, 5, 4\} \) into bins of capacity 10.

![Figure 2-1. Optimal bin packing [9]](image)

BPP was proven to be NP-hard in the strong sense, by reducing the 3-PARTITION problem to the decision version of BPP. The 3-PARTITION problem is NP-complete in the strong sense, which implies that no pseudo-polynomial time algorithm exists if and only if \( P \neq NP \) [5]. Thus, the only known approaches to problems of this nature come in the form of heuristics. Furthermore, the problem of deciding whether \( n \) objects can be packed into no more than \( k \) bins is known as the decision version of the bin-packing problem (DBP).

### 2.2 Open-end Bin Packing

Open-end bin packing was defined in [8] as a variant of the classical bin packing problem. For a given list \( L = \{a_1, a_2, \ldots, a_n\} \) of pieces, where each piece is of size \( s(a_i) \in (0,1] \), the goal is to pack the pieces into as few unit-size bins as necessary. Unlike the classical version, the open bin could accommodate an object extending beyond its boundary, provided that some fraction of the item was within the bin's free space. The extension of the blocks can occur at both ends of a bin, in the one-dimensional case, or an
arbitrary number of faces in higher dimensions. Concretely, suppose we have an item, \( a \), and a bin, \( B \). The item could be packed into \( B \) if \( B \) had free space at one end. The size of the free space must be non-zero (greater than or equal to the highest precision of a machine's decimal representation).

2.3 Bin Packing in Higher Dimensions

In the typical 2-dimensional bin packing optimization problem, objects have to be arranged so that they do not overlap, using as few bins as possible. Figure 2-2 [1] shows bin packing in two dimensions, which allows for freedom in arranging objects inside the bin, making the problem more generalized and complex. Young, et al. [21, 22] proposed the open-end bin packing model.

Figure 2-2. Two-dimensional bin packing.
Some algorithms pack rectangles parallel to the sides of a rectangular bin. This is orthogonal packing without rotation [1], like arranging items on bulletin boards. Likewise, the open-end version of this problem is defined in [15]. As shown in Figure 2-3, it allows blocks to protrude out of one, two, three, or all four sides. The ordinary two-dimensional version is NP-hard [1], and so it follows that the open-end version is just as hard.

![Figure 2-3. Two-dimensional open-end bin packing configurations](image)

Another possible constraint for BPP requires bins to be compact, which aims to minimize gaps between two blocks, or a block and retaining wall. Adding parameters widens the domain in which this problem can be applied. Cases involving more than two dimensions establish even higher levels of generality and complexity.

Young and Li [16] extended open-end bin packing to higher dimensions. Placing objects into the bed of a moving truck subject to the laws of physics would be a practical example. Figure 2-4 shows (a) a configuration with only the front face open; (b) a
configuration with the front and right faces open; and (c) a configuration with the front, right, and top faces open.

![Diagram showing three-dimensional open-end bin packing configurations]

(a) One open face  (b) Two adjacent open faces  (c) Three open faces

**Figure 2-4. Three-dimensional open-end bin packing configurations**

2.4 Best Fit, Next Fit, Random Fit, and Worst Fit

It was proven in [9] that no optimal on-line algorithm exists for the open-end bin-packing problem. Experiments have proven several proposed heuristics efficient in terms of run-time complexity, bounded by an approximation constant. [9]

The *Best Fit (BF)* method works to maintain as many free bins as possible, for as long as it can. Given a falling shape, *BF* places it into a bin that: (1) when able to contain the entire piece, leaves a minimal amount of space at the open end; or else (2) has the least available space. *BF* uses the open-end concept to its advantage, that it can put shapes of arbitrary size into any bin having at least 1 free space. As an example, suppose an incoming piece of size 5 is to be placed into one of four open bins of capacity 10 having 4, 5, 6, and 7 free spaces. *BF* will not select the bin with 5, because it does not leave room for other shapes and isn’t the bin with minimal free space. The preferred bin would be the one with 6 free spaces, because it minimizes the amount of remainder
space. Minimizing remainder space takes precedence over finding minimal available space, which is why the bin with 4 is not chosen either. Although BF does not run within a bounded space constraint, experiments nonetheless have shown this to be perhaps one of the simplest ways of maximizing space savings.

The Worst Fit method is more “greedy” in a way that preference is given to the bin with the most available space. If a non-empty subset of the bins has enough free space to contain the entire incoming piece, Worst Fit chooses the one with the most room. Else, if no bin can contain the piece entirely, the bin with minimal available space is chosen. While not as effective as Best Fit, this performs better on average than Next and Random Fit. Consider the following example, in which a piece of size 5 is to be placed into one of four bins of size 10 having 4, 5, 6, and 7 free spaces. Only the bins with 5, 6, and 7 free spaces can contain the entire piece. The nod goes to the one with 7 since it has the freest space.

The Next Fit (NF) method locates the first available bin and places the piece there. This suggests that the bins have a natural ordering, in which its sequence number \( j = (1, 2, \ldots, n) \) determines priority. Unlike BF, NF runs in bounded space but suffers in space savings [3].

The Random Fit heuristic selects an available bin at random. Its success is entirely dependent on the distribution of the incoming shapes. Thoroughly randomizing the distribution theoretically makes Random Fit equally effective to Next Fit.

2.5 Refined heuristics

By refining the methods above to take the overall compactness of the bins into consideration, Refined Random Fit, Refined Worst Fit, and Refined Best Fit attempt to
mitigate the penalties of poorly chosen placements. Compactness is calculated against percentage filled and compared to a pre-determined threshold value, as in Equation 2-1. If the condition holds, the piece goes into the bin with minimal available space. Otherwise, it is placed according to the respective name-wise scheme: *Refined Random Fit* chooses a bin at random; *Refined Worst Fit* places the piece according to the worst fit; and *Refined Best Fit* places the piece according to the best fit. Refined Best Fit performs better on the average than Best Fit. [13]

\[
\frac{\text{size of the minimum space bin} + \text{size of the piece}}{\text{size of the bin}} - 1 > \delta
\]

Equation 2-1 Test condition for refined heuristics [13]

2.6 Sum-of-Squares in Closed Bin-Packing

A more recent heuristic for on-line closed bin packing in one dimension, *Sum-of-Squares (SS)*, has bounded expected waste over all discrete uniform distributions [4]. Given a finite set of bins, each of fixed size \( b \), *SS* is concerned only with the number of bins, \( N_h \), whose current contents total \( h \) such that \( h < b - 1 \). *SS* chooses which bin should contain incoming item \( x \), to minimize the resulting sum of squares, \( \sum_{h=1}^{b-1} N_h^2 \) [4]. Since the bins are of fixed size, scanning the bins requires constant time per item, and therefore, *SS* runs in linear time.

*SS* excels in terms of space savings. To illustrate, let \( F \) represent any given discrete distribution as an ordered triple (\( b, S, p \)), where \( b \) is a constant representing the size of the bins, \( S \) is the set \( \{s_1, \ldots, s_d\} \) of item sizes within the interval \([1, b-1]\), and \( p \) is a probability vector \((p_1, \ldots, p_d)\) where \( p_i > 0 \) is the probability of item size \( s_i \) and
According to [4], the expected waste of \( SS \) on \( n \) items with probability distribution \( F \) is \( EW_n^{SS}(F) \Theta (EW_n^{OPT}(F)) \). Taken over the union of all discrete uniform distributions \( U[j, k] \), the expected waste of \( SS \) is bounded by a constant factor, \( EW_n^{SS}(U\{j,k\}) O(1) \), ranking \( SS \) among the best heuristics for bin-packing.

Coffman, et al. suggests that, if perfect packing exists for a given instance and the probability of object \( n \) being selected is uniformly distributed, the optimal expected waste for on-line bin-packing algorithms is either \( \Theta(n) \), \( \Theta(n^{1/2}) \), or \( O(1) \). As a result, they stated the following “perfect packing” theorem for one-dimensional on-line bin packing.

For positive integers \( k, j, \) and \( r \), with \( k \neq j \), one can perfectly pack a list \( L \) consisting of \( r \cdot j \) items, \( r \) each of sizes 1 through \( j \), into bins of size \( k \) if and only if the sum of the \( r \cdot j \) item sizes is a multiple of \( k \). [4]

### 2.7 The game of Tetris

Tetris is a computer game invented by mathematician Alexey Pazhitnov in the mid-1980s. One of the most widespread computer games ever created, it was already the best-selling game in the United States and England by 1988. Today, over 50 million copies of the game have been sold worldwide [3]. Today, implementations of Tetris appear all over the Internet. Some of these can be found by pointing a Web browser to

- http://www.liacs.nl/~rbreukel/tetrisai/

To set up the game, we are given an initially-empty game board, which is a rectangular grid of squares. In typical Tetris implementations, the board measures 20
squares by 10 squares. A sequence of tetrominoes (or “tetrads”) as shown in Figure 2-5 is probabilistically generated. The incoming piece appears in the top row of the board and begins falling. As it falls, the player can rotate the piece and move it horizontally. It stops falling when it lands on a filled square, though the player has a last-moment opportunity to slide or rotate it before it stops moving permanently. When all squares in an entire row of the board are filled, it is cleared and all rows above fall one row lower. Then, the top row of the game board is replaced by a new unfilled row. Figure 2-6 shows that multiple rows can be eliminated simultaneously. As soon as a piece is fixed in place, the next piece appears. To assist the player in planning his move, typically a one-piece look-ahead is provided: when the \( i \)th piece begins falling, the identity of the \((i+1)\)th piece is revealed. A player loses when a new piece is blocked by filled grid squares from entirely entering the game board. Pieces continue to be generated until the player loses, meaning that the player can never actually “win” a Tetris game. Thus, the Tetris player’s true objective is to maximize his or her score, until the game is over.

![Figure 2-5. Tetrads](image)

2.8 From the game of Tetris to the TETRIS Problem

The game of Tetris was first generalized into a formal problem by Breukelaar, \textit{et al.} [3] which became known as the TETRIS Problem. For an explanation of the terminology, see Section 3.1. The TETRIS Problem is given below.
The Tetris problem. We will consider a variety of different objectives for Tetris (e.g., maximizing the number of cleared rows, maximizing the number of pieces placed without a loss, etc.) For the decision version of a particular objective $\Phi$, the Tetris problem $\text{Tetris}[\Phi]$ is formally as follows:

**Given:** A Tetris game $G = (B, P_1, P_2, \ldots, P_p)$

**Question:** Does there exist a trajectory sequence $\Sigma$ so that $\Phi(G, \Sigma)$ holds?

The off-line version of TETRIS was proven to be NP-hard in [3] by reducing the 3-PARTITION Problem [5] to the off-line decision version of TETRIS. However, this proof applies to an initially non-empty game board. The problem where the game starts with an empty game board remains open.

Figure 2-6. Falling Tetris piece is about to eliminate the bottom two lines
2.9 Applet Environments

There are a variety of ways to implement simulators, but graphical simulations have distinct advantages. People are "visual creatures" [14], a fact asserted in countless studies on learning [13]. Applets are completely graphical and free-standing programs created using the Java programming language. They are embedded within HTML documents, enabling them to be accessed on any operating system capable of running Java and connecting to the World Wide Web. Microsoft Windows, Linux, and MacOS, to name a few—can load and run them inside Web browsers [7]. Therefore, applets are effective tools for teaching bin-packing using colors and shapes.

Creating a graphical interface, programming the falling blocks, and scheduling real-time object rendering requires days of development. It would involve extensive knowledge of computer graphics, multi-threading, and user-interaction programming—not to mention hours of debugging and refining. The Java 2 Software Development Kit provides all of the libraries needed to implement the simulator. The Java Swing API, introduced with Java 1.2, provides the tools necessary to create the graphical user interface. The Java Threading API, available since the first iteration of Java, allows applications to multi-task, which is an important requirement for simulator applications. Java Threads are simple to manage, yet robust enough to support running complicated tasks in parallel. Finally, we can use Java’s event-trigger model to help threads send messages and status information back and forth. The ubiquitous Java text by Cornell and Horstmann [10] provides complete information on threads and the event-trigger model. Designing Swing applications is discussed in [18] and [23].
2.10 Open Software Standards

A key to designing an open standard is making products extensible. Extensible software frees programmers from having to modify underlying base code in order to satisfy module interdependencies after a change [7]. Coupling [19] is the rate of interdependency between software modules. An example of tight coupling could be a code module that renders graphics and affects program logic simultaneously. One way to avoid coupling is to use or extend self-contained components. In the study of object-oriented programming, cohesion [19] is the measure of a module’s own self-containment. Therefore, the qualities of extensible software are loose coupling and high cohesion.

From a design standpoint, adhering to the Model-View-Controller (MVC) architecture promotes loose coupling and high cohesion. A simulator program can be broken down into model, view, and controller aspects. The model defines the structure of the application, such as the objects, algorithms, and execution logic. The view controls how information is represented externally and visually, as in the geometric interpretation of Tetris and a simulator user interface. A controller facilitates communication between view and model. Because of their threaded nature, applets can be suitable environments for running MVC programs.
CHAPTER 3 DEFINING TETRIS-PACKING

We introduce a formal representation of a class of Tetris-like problems that includes the original Tetris Problem. This introduction uses most of the same notation given in [3] with minor modifications.

This chapter is organized as follows. In the first section, we introduce and define the terminology. The problem description is broken down into these parts: game board, piece state, game rules, rotation model, and player agility model. In the second section, we give some precise notation. In the third section, we formally define the Tetris-Packing Problem (TPP).

In the next chapter, we prove some complexity results of TPP.

3.1 TETRIS-PACKING PROBLEM Terminology

Game board an initially-empty grid of \( m \) rows and \( n \) columns, indexed from bottom-to-top, left-to-right.

Board configuration the state of a game board describing which of its grid squares, if any, are filled.

Game pieces connected polygons composed of an arbitrary number of \( 1 \times 1 \) grid squares. Pieces do not contain holes.

Translation, rotation, drop, fix the only legal moves in Tetris. Translation is moving the falling piece one unit square left or right. Rotation is a 90-degree transformation of the falling piece about its head. In TETRIS-PACKING, the head of a piece is the left-most grid square of its upper-most row. Figure 3-1 shows the heads of the tetrominoes marked with a star symbol. A drop is moving the falling piece one unit down. A fix move stops the falling piece from moving any further.
Rotating pieces a rotation model is a computable function. We will consider the instantaneous rotation model: fix the piece at its head and rotate the piece around that point.

Orientation a piece’s directionality, either 0, 90, 180, or 270 degrees counterclockwise from its base orientation (see below).

Base orientation the initially-defined (non-rotated) configuration of the piece.

Free space a connected region of unoccupied grid squares.

Trajectory a sequence of legal moves consisting of the position-by-position path taken by a piece as it falls, starting at an initial position above the game board and ending when the piece lands. The result of a trajectory is the next game board configuration.

Trajectory sequence given an initial board configuration, an alternating sequence of pieces and trajectories ending with the final board configuration.

Game over condition that arises when the game cannot set the initial configuration of the next incoming piece. The player loses the game at this point.

Placement like dropping objects into a box, the general strategy for the Tetris-Packing Problem is to pack pieces inside the board as compactly as possible. The always-present “gravity” effect aligns pieces on the board’s base or on the top of other pieces. Pieces must remain connected upon landing, even if overhanging parts covers empty space.
**Holes** inaccessible free space concealed from above by filled squares. The sequential nature of the incoming pieces does not permit reversing a placement, and so the accumulation of holes is the unavoidable penalty of a poor placement. In fact, no way\(^1\) exists to decide the overall best placement based on the remainder of the shape sequence.

**Acyclic** objectives do not contain multiple instances of any state (a cycle). Our objective is acyclic because it only depends on the final placement of each piece.

**Checkable** objectives can be verified in time polynomial to the input. TETRIS-PACKING considers the objective of maximizing the number of filled grid squares before a loss. Verification only requires a quick scan of the board with each trajectory, making this checkable as well. Breukelaar, et al [3] define the notion of acyclic checkable objectives and such that Tetris is in NP if the objective is acyclic and checkable.

### 3.2 Notation

The ordered pair \((i, j)\) refers to the grid square at row \(i\), column \(j\) of the game board, which can either be **filled** or **empty**. No completely empty rows can exist under a filled square. The side (top) through which pieces enter is open, allowing pieces to be fixed partially inside the board. Grid squares beyond the other three walls are always filled, acting as sentinels disallowing movement over those boundaries.

Piece “state” is defined as a 4-tuple \((t, \varphi, (i, j), f)\), where

1. A piece type \(t \in \{T_1, T_2, T_3, \ldots, T_x\}\)

---

\(^1\) The game of Tetris allows looking ahead at the next incoming piece. However, the game can be modified to allow viewing the next \(m\) pieces in the sequence. Because the sequence is infinite and random, this is still insufficient for achieving “perfect” packing, unless \(P = NP\).
2. An orientation $\varphi \in \{0, 90, 180, 270\}$ degrees, the amount of counter-clockwise rotation from its base orientation

3. A position $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$ of the piece “head” on the game board, dependent on the piece type

4. The value $f \in \{\text{unfixed}, \text{fixed}\}$ indicating whether the piece can continue to move

Initially, pieces are in base orientation, poised at $\left(\frac{m, \lfloor n / 2 \rfloor}{4}\right)$, and unfixed.

Let $P$ and $P'$ be piece states, $\theta \in \{-90^\circ, 90^\circ\}$ be the rotation angle, and $B$ be a game board configuration. The rotation model is a function $R: (P, \theta, B) \rightarrow P'$. This rotation model adheres to the reasonability standards given in [3] and is comparable to the Tetris rotation model without loss of generality.

1. If $P = (t, \varphi, (i, j), f)$ and the rotation is legal, then $P' = (t, (\varphi + \theta) \mod 360, (i, j), f)$. If the rotation is illegal, then $P' = P$.

2. If the rotation is legal, then $P'$ does not occupy any grid square already filled in $B$.

A free space is a 4-tuple $(s, p, h, d)$ of numbers where $s =$ the size (width) of the space, $p =$ its horizontal location (position), $h =$ its vertical distance (height) from the floor, and $d =$ its hole depth.

**Playing the game.** The following moves are legal for a piece $P = (t, \varphi, (i, j), \text{unfixed})$, with current game board $B$:

1. The new piece state following a rotation is $R(P, \pm 90^\circ, B)$.

2. If the grid squares adjacent to $P$ are open in $B$, then we can translate $P$ left
or right one column. The new piece state is \((t, \varphi, (i, j \pm 1)), \text{unfixed})\).

3. The new piece can drop by one row, if all of the grid squares beneath the piece are open in \(B\). The new piece state is \((t, \varphi, (i - 1, j)), \text{unfixed})\).

4. If at least one grid square immediately below the piece is filled in \(B\), the new piece state is \((t, \varphi, (i, j)), \text{fixed})\).

5. No moves are legal for a piece \(P = (t, \varphi, (i, j)), \text{fixed})\).

The result of a trajectory \(\sigma\), for a piece \(P\) on game board \(B\), is a new game board \(B'\), defined as follows:

1. The new game board \(B'\) is initially \(B\) with the grid squares of \(P\) filled.

2. If the next piece’s initial state is blocked in \(B'\), the game ends and the player loses.

From an initial board configuration \(B_0\) and sequence of pieces \((P_1, \ldots, P_p)\), we use the notation \(\Sigma = (B_0, \sigma_1, B_1, \ldots, \sigma_p, B_p)\) to describe a trajectory sequence. For every piece \(P_i\) on game board \(B_{i-1}\), the trajectory \(\sigma_i\) results in game board \(B_i\). If there is a losing move \(\sigma_q\) for some \(q \leq p\) then \(\Sigma\) terminates at \(B_q\) instead of \(B_p\).

3.3 Definition of Tetris-Packing

Let: \(\beta \in \{\text{empty, arbitrary}\}\) specify the game board’s initial configuration; \((w, h)\) be an ordered pair of positive integers corresponding to the width and height of the board, respectively; \((P_1, P_2, \ldots, P_p)\) be a sequence of \(p\) pieces; \(\{S_1, S_2, \ldots, S_s\}\) be a set of \(s\) shapes, such that for \(1 \leq i \leq p\), \(P_i \in \{S_1, S_2, \ldots, S_s\}\); \(u \in \{\text{open, closed}\}\) signifying whether the top side is open (pieces can partially extend beyond the top) or closed (pieces completely confined). The Tetris-Packing Problem is defined formally as follows.
TETRIS-PACKING PROBLEM (TPP)

**Input:** A tuple $G = (\beta, (w, h), (P_1, P_2, \ldots, P_p), \{S_1, S_2, \ldots, S_s\}, u)$

**Output:** A trajectory sequence $\Sigma$ that optimizes a given objective function.

TPP is a new class of packing problems whose complexity is interesting to study. We state some NP-hardness results in Chapter 4. The similarities between 2-D On-line Open-end Bin Packing and the Open-end Tetris-Packing Problem, or TPP[empty, $(w, h)$, $(P_1, P_2, \ldots, P_p), \{S_1, S_2, \ldots, S_s\}, \text{open}$], suggest that good bin-packing strategies might also be good choices for Open-end TPP. We propose some heuristics for Open-end TPP in Chapter 5. Finally, in Chapter 6, we evaluate their effectiveness in terms of their ability to maximize filled grid squares. Table 3-1 compares Open-end Bin Packing to Tetris Packing.

**Table 3-1. Comparison of Bin Packing and Tetris Packing**

<table>
<thead>
<tr>
<th></th>
<th>Bin Packing</th>
<th>Tetris Packing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation</td>
<td>Arbitrary</td>
<td>Counter-clockwise in increments of 90 degrees</td>
</tr>
<tr>
<td>Typical objective functions</td>
<td>Number of bins, pieces packed</td>
<td>Filled area, pieces packed, complete rows, multi-row completions</td>
</tr>
<tr>
<td>Piece positioning</td>
<td>Anywhere inside the bin</td>
<td>Gravitational effect forces pieces to be bottom-aligned; must consider valid trajectories</td>
</tr>
</tbody>
</table>

21
CHAPTER 4 COMPLEXITY OF TETRIS-PACKING

Intuitively, TETRIS-PACKING is a complex problem. This chapter is devoted to analyzing its complexity.

This chapter is organized as follows. In the first section, we review some known results and give a non-deterministic algorithm for TPP. In the second section, we show that despite starting with an empty game board and placing limits on the game board’s size and shape set, TPP remains NP-hard. The third section summarizes all of the complexity results. Note that every trajectory sequence is feasible when using an empty initial configuration, which implies that every objective function is acyclic and checkable. Therefore, analysis of the existence of such a trajectory sequence can be omitted in all proofs.

In the next chapter, we propose four heuristics for TPP, examine their internal mechanisms systematically, and present a complexity analysis.

4.1 Known Results

The NP-hardness results of this chapter are reductions from the PARTITION Problem to the decision version of the problems.

PARTITION

Instance: A set of integers \( A = \{a_1, a_2, \ldots, a_n\} \)

Question: Is there a subset \( A' \) of \( A \) such that \( \sum_{a \in A'} a < \sum_{a \in A-A'} a \)?

4.2 NP-hardness Results

This section begins with the definition of the decision version of TPP.
DECISION TETRIS-PACKING PROBLEM (DTPP)

**Instance:** A tuple $G = (\beta, (w, h), (P_1, P_2, \ldots, P_p), \{S_1, S_2, \ldots, S_s\}, k, u)$, where $k$ is an integer, $0 \leq k \leq hw$.

**Question:** Does a trajectory sequence $\Sigma$ exist that fills at least $k$ grid squares?

Clearly, a polynomial-time algorithm for DTPP exists whenever $k = 0$. On the other hand, $k = hw$ is a requirement for complete packing.

4.2.1 Closed-end TPP

The next results have to do with a closed game board.

**Theorem 1.** TPP[empty, (arbitrary $w$, $h = 1$), (P_1, P_2, \ldots, P_p), \{arbitrary\}, closed] can be solved in polynomial time.

**Proof.** This closed, unit-height TPP variant involves a bin, closed above, whose height is one unit and width is $w$ units. Pieces have to be rotated so that they are one unit tall, and then inserted towards the left-hand side of the free area, as shown in Figure 4-1. Incoming shapes whose height and width are both greater than one unit automatically terminate the sequence and force the algorithm to return. We give a formal polynomial-time algorithm in Figure 4-2.

![Figure 4-1. Optimal solution to Closed TPP, h=1](image)
Closed_Unit_Height_Rectangle_Packing(w, (P_1, P_2, ..., P_p))
rem = w, sum = 0
begin
    for each P in (P_1, P_2, ..., P_p) loop
        if width(P) > height(P) then
            rotate(P)
            if height(P) > 1 then
                return 0
            else
                if width(P) > rem then
                    return sum
                else
                    let sum = sum + width(P_i)
                    let rem = rem - width(P)
            end
        end loop
    return sum
end

Figure 4-2. Algorithm for Closed TPP with height 1

Analysis. The loop runs up to p times. Determining width and height of a piece takes O(1) time. Placement and rotation take O(1) time as well. Therefore, CLOSED_UNIT_HEIGHT_RECTANGLE_PACKING(w, (P_1, P_2, ..., P_p)) runs in O(p) time. □

Theorem 2. TPP[empty, (arbitrary w, h = 2), (P_1, P_2, ..., P_p), \{rectangles\}, closed] is NP-hard.

Proof. We will show that the decision version of this problem, DTPP[empty, (arbitrary w, h = 2), (P_1, P_2, ..., P_p), \{rectangles\}, k, closed] is NP-complete.

TPP is in NP

Guess a trajectory sequence, \( \Sigma \), and confirm that enough grid squares have been filled. Guessing a trajectory sequence and counting filled squares can both be done in polynomial time.
PARTITION $\preceq$ DTPP

Given an arbitrary instance of PARTITION, i.e. $A = \{a_1, a_2, \ldots, a_n\}$, construct an instance of DTPP in the following manner. Without loss of generality, assume that $a_1 + a_2 + \ldots + a_n = 2B$. See Figure 4-3.

**Game board:**

Let $(w, h) = (101B, 2)$.

**First piece: 1st enforcer tetrad**

Construct $P_1$ with dimensions $100B \times 1$ ($100B$ units wide $\times$ 1 unit tall)

**2nd to (n+1)th pieces:**

For each $a_i$, construct the partition-piece $P_{i+1}$ with dimensions $a_i \times 1$.

**(n+2)th piece: Second enforcer tetrad**

Construct $P_{n+2}$ with dimensions $100B \times 1$

**Shape set:**

Let $\{S_1, S_2, \ldots, S_s\}$ be the set of all shapes in $(P_1, P_2, \ldots, P_{n+2})$

**Threshold:**

Let $k = 202B$.

This transformation can be done in polynomial time based on $p$. 

25
Figure 4-3. Reduction from PARTITION to Closed TPP, \( h = 2 \)

\[ \text{First enforcer piece:} \quad P_1 \]

\[ \text{Partition piece for} \ a_i: \quad P_{i+1} \]

\[ \text{Second enforcer piece:} \quad P_{n+2} \]

\( (\Rightarrow) \) The partition-pieces drop in order by number, as shown in Figure 4-4. First, \( P_1 \) drops into the bottom row, adjusted towards the right end. Then, for each \( P_i, 2 \leq i \leq n + 1 \): if \( a_{i-1} \) belongs in the partition set \( A' \) then \( P_i \) is dropped into the region labeled Area 1, right-adjusted; else, \( a_{i-1} \) does not belong in \( A' \) and \( P_i \) is dropped into the region labeled Area 2, right-adjusted. Finally, \( P_{n+2} \) drops into the remaining space in the second row. Note that the entire board (consisting of 202\( B \) grid squares) is filled using this method.

Figure 4-4. A "yes" instance of PARTITION.

\( (\Leftarrow) \) The reduction arranges the \( n + 2 \) pieces as shown in Figure 4-5. Since \( P_1 \) and \( P_{n+2} \) are each of width 100\( B \) and height 1, the region labeled \( P' \) occupies exactly 101\( B \) – 100\( B = B \) grid squares.

26
If we reposition the regions to resemble the configuration in Figure 4-4, we can obtain the partition set $A' = \{a_j \mid P_{j+1} \in P'\}$. □

Theorem 2 can be generalized for closed game boards of fixed height greater than 1. Generally speaking, therefore, closed TPP is NP-hard. We present the following corollary to Theorem 2.

**Corollary 1.** TPP[empty, (arbitrary $w$, fixed $h \geq 2$), ($P_1$, $P_2$, … , $P_p$), \{rectangles\}, closed] is NP-hard.

*Proof.* This is an extension of the proof of Theorem 2. We append $(h - 2)$ enforcer pieces measuring $1 \times 101B$ to the piece sequence, in order to occupy the space in the $(h - 2)$ rows above the bottom two rows. Let $k = 101Bh$. The entire reduction is still polynomial-time feasible. See Figure 4-6. □
4.2.2 Open-end TPP

The next results reveal the intrinsic complexity of open-end TPP.

**Theorem 3.** TPP[empty, (arbitrary \( w, h = 1 \)), (P_1, P_2, \ldots, P_p), \{rectangles\}, open] is NP-hard.

**Proof.** We will show that the decision version of this problem, DTPP[empty, (arbitrary \( w, h = 1 \)), (P_1, P_2, \ldots, P_p), \{rectangles\}, \( k \), open], is NP-complete.

TPP is in NP

As stated in the proof of Theorem 2, guess a trajectory sequence, \( \Sigma \), and confirm that enough grid squares have been filled. Guessing a trajectory sequence and counting filled squares can both be done in polynomial time.

**PARTITION \( \propto \) DTPP**

Given an arbitrary instance of PARTITION, \( i.e. \ A = \{a_1, a_2, \ldots, a_n\} \), construct an instance of DTPP in the following manner. Without loss of generality, assume that \( a_1 + a_2 + \ldots + a_n = 2B \). Further, let \( C \) be an arbitrary integer, such that \( C > 1 \). See Figure 4-7.
**Game board:**

Let \((w, h) = (Cn + B, 1)\).

**Pieces **\(P_1\) to \(P_n\):

For each \(a_i\), construct the partition-piece \(P_i\) with dimensions \((C + a_i) \times C\).

**Shape set:**

Let \(\{S_1, S_2, \ldots, S_s\}\) be the set of all shapes in \((P_1, P_2, \ldots, P_{n+2})\).

**Threshold:**

Let \(k = Cn + B\).

This transformation can be done in polynomial time based on \(p\).

\((\Rightarrow)\) The partition-pieces are placed according to Figure 4-8. For every \(P_i, 1 \leq i \leq n:\)

- if \(a_i \in A'\) then \(P_i\) is placed horizontally and right-adjusted, rotating when necessary;
- else, \(a_i \not\in A'\) and \(P_i\) is placed vertically and left-adjusted (again, rotating when necessary).

Note that all pieces can only be placed on the bottom row, never on top of another piece, because carefully choosing a value for \(C\) ensures that each partition-piece is at least as high as the game board.

**Figure 4-7. Reduction from PARTITION to Open-end TPP, h=1, rectangles**
Figure 4-8. Another "yes" instance of PARTITION.

\((\leq)\) The reduction arranges the \(n\) partition-pieces according to Figure 4-9 below. It is not necessary for the vertical pieces to be grouped together, nor is it necessary for the horizontal pieces to be grouped together. For illustrative purposes, the piece sequence shown consists of four pieces \((P_1, P_2, P_3, P_4)\).

Figure 4-9. Another "yes" instance of TPP.

Each partition piece contributes \(C\) width (total \(Cn\)), regardless of orientation. Those placed horizontally contribute additional width summing to \(B\). From this instance of TPP, therefore, we can obtain the partition set \(A' = \{a_j | P_j \text{ is horizontal}\}\). □

As with the closed case, we now show that the open-end problem is NP-hard for fixed height \(h \geq 2\) as well.
Corollary 2. TPP[empty, (arbitrary w, fixed \( h \geq 2 \)), (P_1, P_2, \ldots, P_p), {rectangles}, open] is NP-hard.

Proof. The proof is carried out in the same manner as the proof of Theorem 3, but with the following modifications to the reduction.

\[ C = \max\{B, h\} \]

\[ k = (Cn + B) \cdot h \]

Note that the shapes are created using this new value for \( C \) and that the height of the game board is now \( h \). Carefully choosing a value for \( C \) ensures that partition-pieces can only be placed on the bottom row, by making the height of each partition-piece to be at least \( h \). The proof is similar to that of Theorem 3. \( \square \)

4.3 Summary of Complexity Results

We summarize the complexity results stated thus far, in Table 4-1.
<table>
<thead>
<tr>
<th>Board init. config.</th>
<th>Board size ((w, h))</th>
<th>Piece set</th>
<th>Top</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty ((arb \ w, h = 1))</td>
<td>rectangles</td>
<td>Closed</td>
<td>(O(p)) [Implied by Theorem 1]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, \text{fixed} \ h \geq 2))</td>
<td>rectangles</td>
<td>Closed</td>
<td>NP-hard [Corollary 1]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, h = 1))</td>
<td>arbitrary</td>
<td>Closed</td>
<td>(O(p)) [Theorem 1]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, \text{fixed} \ h \geq 2))</td>
<td>arbitrary</td>
<td>Closed</td>
<td>NP-hard [Implied by Corollary 1]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, h = 1))</td>
<td>rectangles</td>
<td>Open</td>
<td>NP-hard [Theorem 3]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, \text{fixed} \ h \geq 1))</td>
<td>rectangles</td>
<td>Open</td>
<td>NP-hard [Corollary 2]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, h = 1))</td>
<td>arbitrary</td>
<td>Open</td>
<td>NP-hard [Implied by Theorem 3]</td>
<td></td>
</tr>
<tr>
<td>Empty ((arb \ w, \text{fixed} \ h \geq 1))</td>
<td>arbitrary</td>
<td>Open</td>
<td>NP-hard [Implied by Corollary 2]</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 5 HEURISTICS FOR TETRIS-PACKING

We present four new on-line Tetris-packing heuristics: Corner Fit, Deepest Fit, Hybrid Corner/Deepest Fit, and Modified Best Fit. Although we designed these algorithms for two dimensions, they can easily extend to higher dimensions. Recall that a free space is a 4-tuple \((s, p, d, h)\) corresponding to size, position, depth, and height, respectively.

This chapter is organized as follows. Each of the first five sections describes a heuristic and contains a pseudocode implementation. Section 5.6 contains a complexity analysis and comparison.

5.1 Heuristic I - Corner Fit

The Corner Fit (CF) heuristic places a block into the lowest available corner. The idea behind CF is to prevent covering small pockets with large blocks (Figure 5-1). More formally, given any two free spaces \(F_1 = (s_1, p_1, d_1, h_1)\) and \(F_2 = (s_2, p_2, d_2, h_2)\), CF will drop the falling piece into \(F_1\) only if \(s_1 \geq s\) and \(s_1 \leq s_2\). Otherwise, CF will choose \(F_2\). Often the chosen free space is larger than the falling piece. If this occurs, the shape will go at the left end or right end of the free space, depending on whether the space lies mostly on the left or right half of the bin, respectively. This is the “left-right” logic used by CF.
Corner Fit does a decent job of filling free spaces and avoiding waste near the bottom of the bin. It has a slight weakness, though, in that deep valleys tend to accumulate in the center of the bin, a behavior intrinsic to the left-right logic on which it relies heavily. This becomes an issue when we attempt to drop a wider piece in the presence of a singular valley one space wide. Until a shape of unit width arrives, CF keeps moving shapes far left (or right). Statistically, if there is only one shape of unit width and several shapes having non-unit width, chances are that the valley will grow deep.

The GET_FREE_SPACES_FOR function (Figure 5-2) is the key to these heuristics.
function GET_FREE_SPACES_FOR(falling_shape)
TAKEN array of boolean{ false }
Let rowNum = falling_shape.y + 1
while rowNum < binHeight do
  let height = binHeight - rowNum + 1
  for c in 1 to binWidth
    if !TAKEN[c] and bin[rowNum][c] is NULL then
      let TAKEN[c] = true
      let blankCount = blankCount - 1
    end if
  end for
  if blankCount > 0 then
    let w = 0
    for x in 1 to binWidth
      if not TAKEN[x] then
        let w = w + 1;
      else
        if w >= minWidth then
          addSpace(w, x-w, height, depth);
        end if
        let w = 0
      end if
    end for
    if w >= minWidth then
      addSpace(w, taken.length-w, height, depth));
    end if
    exit while
  end if
end while
return SPACES
end function

Figure 5-2. GET_FREE_SPACES_FOR algorithm

To understand how searching for free spaces works, consider the example of a closed 10 × 10 bin partially filled, in Figure 5-3.
Figure 5-3. Operation of Corner Fit

Rotation the shape onto a flat, horizontal position. This step is designed to avoid adding height to the stack too quickly.

The free space scan begins with the row directly beneath the falling piece. There are no units occupied, so add one free space F1 having coordinates p1 = 1, s1 = 10, h1 = 6, d1 = 1.

Three spaces are taken at positions 1, 2, and 3. Mark the first three elements of the TAKEN array and add F2 = (4, 7, 5, 2) to SPACES. Free spaces with lesser size (width) go ahead of other spaces in the SPACES set, thus F2 goes in front of F1.

The same three spots are taken. No changes are made to TAKEN. Add F3 = (4, 7, 4, 3) in front of F2 because h3 < h2. (Note: marks from TAKEN are never removed because the marked spaces remain unreachable.)

No changes to TAKEN. Add F4 = (4, 8, 4, 4) in front of F3 because h4 < h3.

The fourth square is taken. The free space is too narrow to contain the piece of width 3. Stop searching here and take the first on the list, F4.
5.2 Heuristic II - Deepest Fit

The *Deepest Fit* (*DF*) heuristic attempts to place the block in a position that will cause the least increase in the height of the stack, as shown in Figure 5-4, by selecting the deepest space. More formally, given two free spaces \( F_1 = (s_1, p_1, d_1, h_1) \) and \( F_2 = (s_2, p_2, d_2, h_2) \), *DF* chooses \( F_1 \) only if \( d_1 \geq d_2 \). Otherwise, it chooses \( F_2 \). The pseudocode is given in Figure 5-4.

```
DEEPEST_FIT()
  if height(shape) < width(shape) then
    rotate(shape)
  end
  let spaces = GET_FREE_SPACES_FOR(shape)
  let selection = SPACES’LAST
  move_shape(selection)
end function

function COMPARE(F1, F2)
  if depth(F1) < depth(F2) then
    return F1
  else
    return F2
  end if
end function
```

*Figure 5-4. Deepest Fit*

Taking after the *Corner Fit* example, Figure 5-5 illustrates searching the free spaces using the *DF* comparison method. The algorithm starts by ensuring that the shape is upright or vertically long. This move prepares the block to fill narrow gaps, like the one at the bottom of the bin.
Add $F_1 = (1, 10, 6, 1)$ to SPACES.

Mark spaces 1, 2, and 3 as TAKEN and add $F_2 = (4, 7, 5, 2)$ to SPACES before $F_1$, because $d_2 < d_1$.

No new spaces occupied. Add $F_3 = (4, 7, 5, 3)$ behind $F_2$.

Found two free spaces, $F_4$ and $F_5$. The order in which they appear relative to each other is arbitrary, since $d_4 = d_5$.

Row scan reveals three more filled squares. Add $F_6 = (2, 5, 5, 5)$.

$F_6$ is added before $F_7$, because $d_6 < d_7$ and therefore $F_6$ precedes $F_7$. No more rows to scan.

$DF$ picks up the last entry in the SPACES list, $F_7$.

Figure 5-5. Operation of Deepest Fit
5.3 Heuristic III - Hybrid Corner/Deepest Fit

Using statistical methods to coordinate use of the corner fit and deepest fit logic, the idea behind the Hybrid Corner/Deepest Fit (HC/DF) strategy is to take advantage of the compactness of CF while covering the deep pockets it leaves behind. Initially, HC/DF runs in CF mode. It computes the height, $h$, of each column, and if $h$ is less than one standard deviation from the average column height, $\bar{h} - \sigma_h$, it triggers DF logic to fill it in. The hybrid runs in DF mode until no deep pockets remain, at which time CF logic resumes control. In Figure 5-6, the algorithm will switch back to the Corner Fit strategy, after filling a deep pocket with Deepest Fit.

```
HYBRID_CORNER_DEEPEST_FIT()
for i in 1 to binWidth
    heights(i) = heightOfColumn(i)
end for
let avgHt = average(heights)
let stdHt = stdev(heights)
for i in 1 to binWidth
    if heights(i) < avgHt - stdHt then
        DEEPEST_FIT()
        return
    end if
end for
CORNER_FIT()
end function
```

Figure 5-6. Hybrid Corner/Deepest Fit

5.4 Heuristic IV: Modified Best Fit

We modified the Best Fit heuristic for Open-end Bin Packing [12] for TPP. The idea behind Modified Best Fit takes advantage of the open-end aspect of the game. Figure 5-7 shows Modified Best Fit dropping a shape into the best-fit free space.
MODIFIED_BEST_FIT()
let i = 0
let spaces = GET_FREE_SPACES_FOR(shape)
while (i < 4 AND empty(spaces))
    rotate(shape)
    let spaces = GET_FREE_SPACES_FOR(shape)
    let i = i + 1
end while
if (empty(spaces))
    move shape all the way to left
else
    move_shape(spaces’last)
end

Function COMPARE(F1, F2)
let fsh = shape’height
if size(F1) > fsh and Size(F2) > fsh then
    If size(F1) <= size(F2) then return F1
    Else return F2
else
    if size(F1) > fsh then return F1
    else return F2
    if size(F1) < fsh then return F2
end

Figure 5-7. Modified Best Fit

Clearly, the Modified Best Fit runs in polynomial time. The
GET_FREE_SPACES_FOR function is the same as the one used in the other heuristics.
However, this heuristic takes better advantage of shape rotation, a practice not shared by
the others. The interested reader may wish to consult the paper on open-end bin packing
[12] for the complete definition of a “best-fit”.

5.5 Baseline Heuristic: Next Fit

We chose Next Fit (Figure 5-8) as the baseline for our performance comparison.

NF is simple and performs acceptably well for on-line open-end bin packing.
5.6 Complexity of Tetris-Packing Heuristics

In order for heuristics to be useful, they must be effective and efficient. Effective heuristics pack bins consistently, while efficient heuristics make decisions in polynomial time. Each algorithm consists of two stages. The first is a data-gathering stage, in which the algorithm queries the state of the game. The state of a game consists of how many free spaces exist that can contain the falling shape, the height of the columns, and percentage filled. The second stage is a decision phase, in which the shape is positioned based on the gathered data. The overall performance, represented in terms of the size of the bin, is equal to the worst-case performance of the two phases.

The data-gathering phases of $CF$ and $DF$ occur in the GET_FREE_SPACES function. The outer-most loop runs in time $O(h)$, and the first nested loop runs in time $O(w)$. If there are no blank spaces in the current row (namely, BLANKCOUNT = 0) then the search for more free spaces terminates, because no falling shapes can be placed on or below that row (unless a new rule was added that allowed “breaking” blocks). The second nested loop runs in $O(w)$. ADDSPACE depends on the implementation of the Collection to which the free spaces are being stored. Most dynamic linear structures in the Java API, such as ArrayList and Vector, implement add() in amortized
constant\(^2\) time [11]. This implementation opts for the Vector, and therefore this second loop runs in \(O(w)\). From this, the GET_FREE_SPACES_FOR function runs in time \(O(hw)\).

Looking at Corner Fit, we see that only one data-gathering step takes place, contributing \(O(hw)\) time. The movement phase can calculate the target using the following algorithm:

```plaintext
compute midpoint of SPACES'FIRST
if midpoint < binWidth / 2 then
  compute POS(SPACES'FIRST) - POS(falling_shape)
else
  compute POS(SPACES'FIRST) + SIZE(SPACES'FIRST) - POS(falling_shape) - SIZE (falling_shape)
```

Figure 5-9. Computing number of squares to slide horizontally

The most direct method of computing SPACES'FIRST is to use a linear-time selection algorithm. The movement phase of CF contributes \(O(1)\). Therefore, Corner Fit runs in \(O(hw)\).

Deepest Fit is similar to Corner Fit in its design, with only two differences: the comparison method used for the free spaces in the SPACES collection, and choosing SPACES'LAST instead of SPACES'FIRST. Comparison still uses constant time, the selection algorithm remains the same, and SPACES'LAST and SPACES'FIRST perform identically. Therefore, Deepest Fit runs on par with Corner Fit, in \(O(hw)\).

In contrast, Hybrid Corner/Deep Fit takes a little more time in the data-gathering phase. It calculates the average and standard deviation of the column heights. Getting the height of a column in a two-dimensional array structure takes \(O(h)\) time, and iterating

\[\text{Figure 5-9. Computing number of squares to slide horizontally}\]
over $w$ columns contributes $O(hw)$ time. The column heights are stored in a temporary array of size $w$ for the calculation of mean and standard deviation, which are $O(w)$ operations. These happen in series, and therefore contribute $O(hw)$ time. The last step of the determination compares the column heights to the average column heights minus one standard deviation and requires $O(w)$ time. Therefore, the overall time needed in deciding whether to use $CF$ or $DF$ is $O(hw)$. The time needed after choosing either option remains $O(hw)$.

Overall, the four proposed heuristics run in $O(hw)$ time. Table 5-1 shows the breakdown by heuristic.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next Fit</td>
<td>$O(hw)$</td>
</tr>
<tr>
<td>Modified Best Fit</td>
<td>$O(hw)$</td>
</tr>
<tr>
<td>Corner Fit</td>
<td>$O(hw)$</td>
</tr>
<tr>
<td>Deepest Fit</td>
<td>$O(hw)$</td>
</tr>
<tr>
<td>Hybrid Corner/Deep Fit</td>
<td>$O(hw)$</td>
</tr>
</tbody>
</table>
CHAPTER 6 T-PACK: SOFTWARE FOR TETRIS PACKING SIMULATION

T-PACK is a complete software package for Tetris-Packing, which includes an application programming interface (API) for implementing and evaluating Tetris-Packing heuristics. This API is written for Java.

This chapter is organized as follows. The first section describes the T-PACK API and how it can be used to implement, display, and evaluate heuristics. The second section contains a description of an on-line applet-based simulator with support for human-computer interaction. In the third section, the off-line benchmarking utility used to analyze multiple heuristics is described.

6.1 T-PACK Application Programming Interface

An application programming interface (API) was created to allow others to implement heuristics of their own design. These heuristics can be uploaded to a centralized server and tested. Algorithms built using the API extend the abstract class PackingAlgorithm, which implements the ChangeListener interface and are always registered as listeners to the game. When a heuristic object receives aChangeEvent from the game, stateChanged() is invoked, during which control is passed to the specific heuristic that computes the best move and requests the game thread to move or rotate the shape. Example code can be found in the accompanying CD-ROM.

6.2 STACK-N-PACK, the game

We developed an interactive Java game, STACK-N-PACK, to visualize Tetris-Packing heuristics. It reads its startup parameters, initializes the board, loads the shape set, and drops shapes one by one into the board. Each shape is positioned into a staging area several units directly above the board and falls at a rate of about one unit square per
second, coming to rest on top of another shape or the bottom wall. Then, the next shape is released.

![Screenshot of the STACK-N-PACK Game](image)

**Figure 6-1. Screenshot of the STACK-N-PACK Game**

The game runs on a feedback system designed around the Java Swing and Abstract Window Toolkit (AWT) event models. It broadcasts a `ChangeEvent` (from the `javax.swing.event` package) to the game’s event listeners, every time a shape is released. Figure 6-2 illustrates the general operation of the game engine.
6.3 Off-line Benchmark Utility

The off-line component is a batch-mode console application that operates much like the game, minus the graphical interface. It allows parameterized tests against the variables listed in the table above and consists of a statistics module, master controller, and player simulator. Test parameters are passed to the main program via command-line arguments. The statistics module constantly logs the state of the simulation, in addition to evaluating a heuristic's performance and efficiency according to the average filling percentage attained. Unlike the game, however, the benchmark utility does not incorporate level-climbing. Each trial stops as soon as the board is filled up. The T-PACK benchmark manual is included in Appendix B.
CHAPTER 7 EVALUATION OF HEURISTICS USING T-PACK

A comprehensive and accurate assessment of the effectiveness of Corner Fit (CF), Deepest Fit (DF), Modified Best Fit (MBF), and Hybrid Corner/Deepest Fit (HC/DF) would involve three or four independent variables. Then, test cases would have to be designed for all possible vectors of those variables. This study, however, consists of two carefully-designed experiments.

This chapter is organized according to the following. In the first section, we describe the method used to conduct our experiments. In the second section, we present the first experiment, using variable bin sizes and nine STACK-N-PACK shapes. The third section presents the second experiment, using a fixed-size bin and the seven Tetris shapes. The chapter ends with charts of the data collected from our experiments and a competitive analysis of the heuristics (proposed and baseline).

7.1 Experimental Method

The simulator runs one heuristic at a time, with an initially-empty game board. During a single trial, an endless supply of pieces, chosen randomly from the shape set, is dropped. The trial ends when an incoming piece cannot be placed. After each piece settles into its resting position, the fill percentage is calculated as the ratio of filled grid squares to the size of the game board (product of width and height). Although all scenarios utilize the open-end top, unit squares within the staging area are not counted in the fill percentage. The number of required trials was determined experimentally, by graphing the average fill percentage gained through Modified Best Fit versus the number of trials performed. We found that the curve flattened out after 500 trials.
Experimental Determination of Required Number of Trials

![Graph showing the best fit percentage_fill average over different trial numbers (n) for various percentiles.]

Figure 7-1. Experimentally determining the number of trials to run per heuristic.

7.2 Experiment 1: variable bin width, nine STACK-N-PACK rectangles

This was perhaps the easiest scenario. In this scenario, all shapes were rectangular, and no shape was more than two units wide. This made stacking easy and reduced the risk for overhanging. The Next Fit (NF) heuristic from Open-end Bin Packing was adapted and used as the baseline. The complete test data is presented in Appendix A.

The baseline heuristic was effective on our STACK-N-PACK simulation. It performed better (fill percentage about 82%) with shorter bins than taller ones (80%). The width of the board was a marginal performance factor, with a difference of only 2 percentage points between the 10-unit and 18 unit-wide bins.
Figure 7-2. Average Performance of Next-Fit

Figure 7-3. Average performance of Corner Fit
Figure 7-4. Average performance of Deepest Fit

Figure 7-5. Average performance of Modified Best Fit
The proposed heuristics performed clearly outperformed NF. Corner Fit maximized its potential with the largest board size; usually exceeding 88% filled area. Deepest Fit outshined NF in just about every test category but fell short of CF. DF made significantly greater strides with large boards. DF had very consistent scores per run. BF was the runner-up in filled area only to CF. In contrast, the performance of the Hybrid Corner/Deepest Fit was surprisingly lower than that of Corner Fit. Theoretically, this heuristic should have performed better than CF because the hybrid is merely an extension of it. During a visual demonstration, the hybrid tried to position a block of width 2 in a free space of width 1. The default action, in this case, is to position the block in the corner of a free space having width greater than 1. This often covers unit free spaces, creating holes. When these holes accumulate, it is not hard to imagine the final fill percentage just missing the threshold. Perhaps some refinement of the switching
A mechanism would be desirable. Figure 7-7 and Table 7-1 summarize the results over all game board sizes.

Figure 7-7. Comparing Overall Average Filled Area Achieved (using rectangles)

Table 7-1. Performance comparison data (rectangular Pieces)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Fill %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified Best Fit</td>
<td>83.2%</td>
</tr>
<tr>
<td><strong>Corner Fit</strong></td>
<td><strong>85.1%</strong></td>
</tr>
<tr>
<td>Deepest Fit</td>
<td>81.4%</td>
</tr>
<tr>
<td>Hybrid Corner/Deepest Fit</td>
<td>82.0%</td>
</tr>
<tr>
<td>Next Fit</td>
<td>81.4%</td>
</tr>
<tr>
<td><strong>Overall Average</strong></td>
<td><strong>82.6%</strong></td>
</tr>
</tbody>
</table>
7.3 Experiment 2: 18 rows x 10 columns, seven Tetris shapes

This case describes the traditional Tetris setup, minus the removal of complete lines. The rectangles from the first scenario were swapped with the set of seven Tetris pieces. The simulation was conducted in the same fashion, but the scores were noticeably lower, probably due to the geometric irregularity of the tetrads, especially the “Z” and “L” pieces.

In contrast to its performance with rectangular shapes, \( NF \) averaged slightly below 60\% with tetrads. In observing the test runs, one key note is that pieces were not spread out over the width of the bin, but instead “congregated” near the midline. Again, \( NF \) was at the bottom. Modified Best Fit fared significantly better than \( NF \). \( CF \) was again the strongest heuristic. The only observed weakness in \( CF \) was not being able to avoid leaving small 1- or 2-unit holes with every piece placed. Since tetrads all have 4 unit elements, a 2-unit gap incurs a hefty penalty, because 6 spaces are being used while receiving credit for 4. Worse, covered empty spaces cannot be reclaimed, according to Tetris packing rules. This can dramatically reduce the efficiency of the heuristic. A summary of the results of this experiment is given in Figure 7-8 and Table 7-2.
Average Fill Percentage - Tetrominoes (500 reps.)

![Bar chart showing fill percentages for different Tetrominoes algorithms.]

Figure 7-8. Average T-PACK scores using tetrominoes

Table 7-2 Performance Comparison Data (tetrominoes)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Fill%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next Fit</td>
<td>59.6%</td>
</tr>
<tr>
<td>Best Fit</td>
<td>74.4%</td>
</tr>
<tr>
<td><strong>Corner Fit</strong></td>
<td><strong>76.2%</strong></td>
</tr>
<tr>
<td>Deepest Fit</td>
<td>73.8%</td>
</tr>
<tr>
<td>Hybrid Corner/Deepest Fit</td>
<td>75.3%</td>
</tr>
<tr>
<td><strong>Overall Average</strong></td>
<td><strong>71.8%</strong></td>
</tr>
</tbody>
</table>

Furthermore, the complexity of the pieces themselves was a limiting factor of the heuristics’ overall fill rates. Figure 7-9 gives a side-by-side comparison of performance under rectangular shapes versus tetrominoes. Overall, the heuristics fared much better with the rectangles.
Figure 7-9. Comparison chart of fill rates using rectangles vs. tetrominoes.
8.1 General remarks about the study

Our brief exploration of the Tetris-Packing Problem (TPP) has covered a broad range of subject areas. We proved that most TPP problems are NP-hard. We also delved into some heuristics for TPP and methods for evaluating them.

The first major result of this research was the creation of the STACK-N-PACK game, whose purpose was to simulate the Tetris-Packing Problem environment and add some entertainment value to the subject area. A Web site was created to complement this new game.

The second major result was a formal specification of a new class of packing problems, called the Tetris-Packing Problem (TPP). This new class is a generalization of the original Tetris Problem, the Open-end Tetris-Packing Problem, and many other variants.

The third major result was the classification of the complexity of TPP variants. We proved that, in the closed TPP, a $O(p)$ algorithm exists for arbitrary shapes and boards having height $h = 1$. We also proved that the closed TPP was NP-hard, for rectangular shapes and boards having fixed height $h \geq 2$. Turning to the open-end case, we proved that open-end TPP was NP-hard for rectangles and boards having fixed height $h \geq 1$. Note that the NP-hardness results for empty initial configuration and objective function of filled grid squares were given here. In contrast, the NP-hardness results for arbitrary initial configurations and objective function of number of lines completed was given in the literature [3].

The fourth major result was the creation of T-PACK, a complete software
package written in Java for designing and evaluating Tetris-Packing heuristics. The entire
package and utilities are free to download, and we hope that this new resource will
promote increased research in TPP.

The fifth major result was composed of the design and evaluation of four
heuristics for open-end TPP. When compared to the baseline heuristic, *Next Fit*, the new
heuristics—Corner Fit, Deepest Fit, Hybrid Corner/Deepest Fit, and Modified Best Fit—
were better in terms of increasing average filled area. Of these four, the Hybrid
Corner/Deepest Fit and Modified Best Fit stood above the rest.

8.2 For further investigation

Virtually all of the accomplishments have inspired some ideas for further
investigation. The STACK-N-PACK game could use more functional, playable, and user-
friendly features. There also ought to be some complexity analysis done on TPP with
other objective functions, such as number of pieces packed, number of complete rows,
*etc*. In particular, for maximizing the number of pieces packed, we have an O(*p*)
algorithm similar to that given in Theorem 1, for TPP[empty, (arbitrary *w*, *h* = 1), (*P*_{1}, *P*_{2},
… , *P*_{*p*}), \{rectangles\}, open].

Because the game board is open-end, pieces can be arbitrarily tall. The greedy
strategy of positioning the pieces vertically in order to minimize their width, is the most
effective. We need only copy the polynomial-time algorithm from Theorem 1 and
remove the conditional check for unit height pieces, obtaining the algorithm in Figure
8-1. The idea behind this modified algorithm can be visualized in Figure 8-2.
Open_Unit_Height_Packing(w) is
begin
    let remain = w, fill = 0
    for i = 1 to p loop
        if width(Pi) > height(Pi) then
            rotate(Pi)
        if width(Pi) > remain then
            return fill
        else
            place Pi left-adjusted
            let fill = fill + width(Pi)
            let remain = remain - width(Pi)
        end loop
    return fill
end

Figure 8-1. Algorithm for Open-end TPP: h=1, rectangles, maximizing pieces packed

Figure 8-2. Optimal solution of Open-end TPP: h=1, maximizing pieces packed

Concerning heuristic development, one direction of further research is improving the implementations of Corner Fit and Deepest Fit. Another direction might be a study on a Hybrid Deepest/Corner Fit (HD/CF) heuristic in which Deepest Fit is the default mode (instead of Corner Fit, as in HC/DF). In addition, creating a T-PACK programming language based on its API, as well as a graphical editing tool, would perhaps decrease the amount of time spent in coding heuristics.
It would be nice to hold a STACK-N-PACK Tournament, allowing heuristic designers to test their work against a database of other heuristics. Similar contests are already in existence, such as the Iterated Prisoner's Dilemma competition (http://www.prisoners-dilemma.com).

8.3 Other implementation challenges

For open-end TPP with multiple open ends (faces), there are the following implementation challenges:

1. Pieces can be dropped through any of the open ends, which complicate the shape trajectory.

2. Assume that we define an “orientation” for each open end, such that each orientation makes its corresponding open end to be considered the “top” of the board. Exactly how orientation-aware should each model element (shape, game board, or player) be?

3. Should there be a logical, orientation-independent coordinate system? How should the mapping of logical to physical (orientation-aware) coordinates be handled?

8.4 An invitation

This thesis evolved from a heuristic programming exercise to an in-depth complexity study of a new class of packing problems. We invite readers interested in the programming aspect to take a look at its complexity side, as well. There is a lot of interesting open research problems in TPP.
REFERENCES


---

3 This compendium was also published in the appendix of G. Ausiello & P. Crescenzi, et al. (1998). Approximate Solution of NP-hard Optimization Problems. Springer-Verlag.


Computation, and Mathematical Concepts. (ERIC Document Reproduction Service No. ED 430 774)


## APPENDIX A EXPERIMENT 1 DATA TABLE (RECTANGLES)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Width</th>
<th>Height</th>
<th>Scr</th>
<th>MinPct</th>
<th>AvgPct</th>
<th>MaxPct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Fit</strong></td>
<td>10</td>
<td>10</td>
<td>137</td>
<td>59%</td>
<td>79%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>192</td>
<td>69%</td>
<td>84%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18</td>
<td>286</td>
<td>77%</td>
<td>87%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>159</td>
<td>61%</td>
<td>79%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>14</td>
<td>233</td>
<td>69%</td>
<td>84%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td>325</td>
<td>75%</td>
<td>87%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>10</td>
<td>226</td>
<td>61%</td>
<td>78%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>326</td>
<td>69%</td>
<td>84%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>18</td>
<td>376</td>
<td>77%</td>
<td>87%</td>
<td>97%</td>
</tr>
<tr>
<td><strong>Corner Fit</strong></td>
<td>10</td>
<td>10</td>
<td>155</td>
<td>61%</td>
<td>80%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>218</td>
<td>71%</td>
<td>85%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18</td>
<td>297</td>
<td>74%</td>
<td>88%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>164</td>
<td>62%</td>
<td>81%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>14</td>
<td>253</td>
<td>73%</td>
<td>86%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td>316</td>
<td>77%</td>
<td>89%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>10</td>
<td>200</td>
<td>62%</td>
<td>82%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>324</td>
<td>69%</td>
<td>87%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>18</td>
<td>432</td>
<td>77%</td>
<td>89%</td>
<td>96%</td>
</tr>
<tr>
<td><strong>Deepest Fit</strong></td>
<td>10</td>
<td>10</td>
<td>152</td>
<td>61%</td>
<td>76%</td>
<td>93%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>211</td>
<td>71%</td>
<td>82%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18</td>
<td>293</td>
<td>78%</td>
<td>86%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>164</td>
<td>59%</td>
<td>75%</td>
<td>93%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>14</td>
<td>258</td>
<td>71%</td>
<td>83%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td>331</td>
<td>77%</td>
<td>86%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>10</td>
<td>178</td>
<td>63%</td>
<td>75%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>287</td>
<td>72%</td>
<td>82%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>18</td>
<td>379</td>
<td>79%</td>
<td>86%</td>
<td>96%</td>
</tr>
<tr>
<td><strong>Hybrid Corner/Deepest Fit</strong></td>
<td>10</td>
<td>10</td>
<td>117</td>
<td>61%</td>
<td>77%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>233</td>
<td>71%</td>
<td>83%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18</td>
<td>276</td>
<td>77%</td>
<td>87%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>172</td>
<td>62%</td>
<td>77%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>14</td>
<td>248</td>
<td>73%</td>
<td>82%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td>330</td>
<td>77%</td>
<td>86%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>10</td>
<td>187</td>
<td>61%</td>
<td>76%</td>
<td>91%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>307</td>
<td>72%</td>
<td>82%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>18</td>
<td>376</td>
<td>78%</td>
<td>86%</td>
<td>96%</td>
</tr>
<tr>
<td><strong>Next Fit</strong></td>
<td>10</td>
<td>10</td>
<td>150</td>
<td>63%</td>
<td>82%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>14</td>
<td>232</td>
<td>59%</td>
<td>81%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>18</td>
<td>272</td>
<td>64%</td>
<td>79%</td>
<td>89%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>10</td>
<td>190</td>
<td>68%</td>
<td>82%</td>
<td>93%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>14</td>
<td>249</td>
<td>70%</td>
<td>82%</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td>323</td>
<td>69%</td>
<td>80%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>10</td>
<td>215</td>
<td>69%</td>
<td>83%</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>270</td>
<td>72%</td>
<td>83%</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>18</td>
<td>363</td>
<td>69%</td>
<td>81%</td>
<td>92%</td>
</tr>
</tbody>
</table>

---

4 Data collected February 20, 2006.
APPENDIX B T-PACK MANUAL

This manual describes the use of the T-PACK offline benchmark.

Syntax

\[ \text{java } -j\text{ar tpack.jar options } -a\text{ Algo1 } -a\text{ Algo2} . . . \]

Parameters:

\[-a \text{ algoname } \ldots\] Loads heuristic named \textit{algoname} which must be a package in the CLASSPATH. Implementations must reside in a class called \textit{Algorithm}.

Options:

\[-s \text{ width [height]} \] Size of the bin, in units. Height requires width. Default width is 10, height is 18.
\[-o \text{ leftOpen [rightOpen]} \] Determines what side of the bin to open. 1 = open, 0 = closed (default)
\[-r \text{ reps} \] Number of times to repeat each algorithm (default = 1)
\[-\text{tetrad} \] Uses tetromino (“tetrad”) shape set

The parameters can appear in any order.

Examples

To run the “Tetris” simulation using Corner Fit only:

\[ \text{java } -j\text{ar tpack.jar } -\text{tetrad } -a\text{ cornerfit} \]

To run a simulation with 500 repetitions, a large board, using Corner Fit and Deepest Fit:

\[ \text{java } -j\text{ar tpack.jar } -s\text{ 18 18 } -r\text{ 500 } -a\text{ cornerfit } -a\text{ deepestfit} \]