Algorithm Visualization

User Guide

Casey Davis Johnathan Mell Di Mu Federico Nusymowicz

{davisca, dimu, jmell, fedenusy} @seas.upenn.edu

University of Pennsylvania

May 4, 2012

1 Instruction Manual

1.1 Overview

This document provides instructions for the basic use of the Algorithm Visualization application, developed for the Android platform. Note that this application was designed primarily for Android tablet devices, so compatibility with smaller screens is not guaranteed.

The second section of this document provides brief technical documentation of the source code for developers wishing to further develop the application or reuse the code.

1.2 Startup

The startup splash screen for the knapsack problem (or more generally, the bin-packing problem) implementation within the Algorithm Visualization application is displayed upon startup (shown in Figure 1). From this screen the user can begin the game, view the current list of high scores, and quit the application.

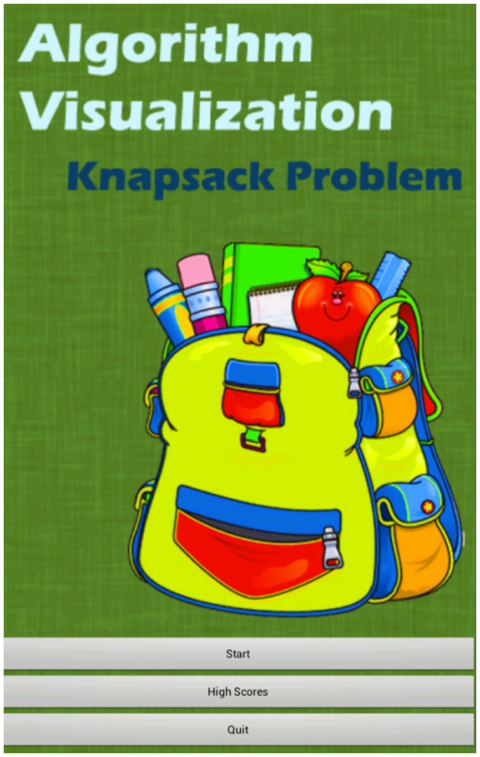

Figure 1: Startup splash screen for the knapsack problem

1.3 High Scores

The High Scores screen (Figure 2) displayes the best time for each level (configured in an accompanying `.xml` file in the source code (see Technical Documentation in Section 2). The 'Reset High Score' button can be used to clear the high score values for all levels, and the 'Return' button takes the user back to the main menu.

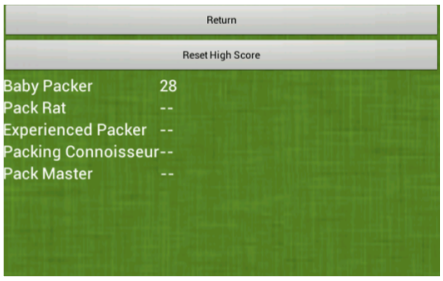

Figure 2: High Scores screen

1.4 Gameplay

Upon beginning a game, a dialog (Figure 3) is displayed that allows the user to begin the game, which starts the timer that runs until the user submits the attempted solution to the problem.

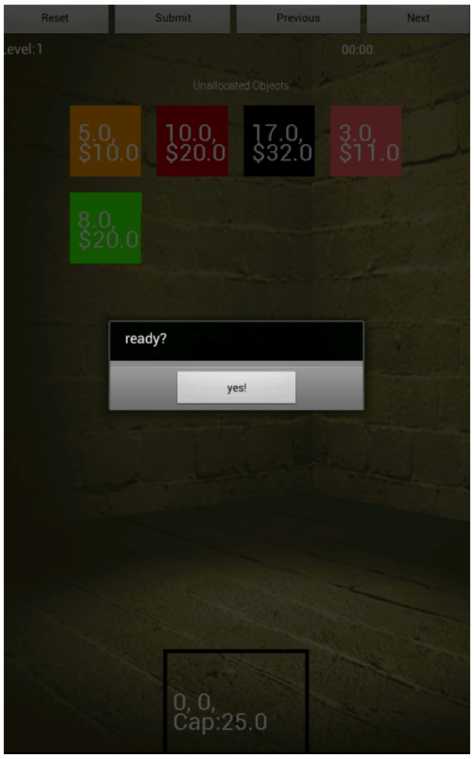

Figure 3: "Ready?" dialog at the beginning of each level

Each level in the game includes a series of objects, displayed at the top of the screen, with associated weights and values. At the bottom of the screen is a series of one, two, or three bins, each with a certain weight capacity, as well as the total value of the objects inside.

The object of the game is to touch and drag the objects into the bin(s) such that the total value of the objects in the bins is maximized, without exceeding the weight capacity of each bin. As objects are added to the bins, the bin will slowly fill to graphically show the remaining capacity of the bin. See Figure 4 for the basic layout of the game.

When the number of objects is large, left and right arrow buttons will allow the user to scroll through a paginated list of objects. The user may also touch the bins to expand a page that displays

Figure 4: Basic layout of gameplay

the objects inside that bin (Figure 5). From here, the objects can be dragged out of the bin entirely, or directly into a different bin. Touching the small 'X' in the upper right-hand corner of the screen will allow the user to return to the page of unallocated objects.

The user can submit the attempted allocation of objects by touching the 'Submit' button at the top of the screen. A dialog will appear indicating whether the solution is optimal, and if not, the approximate percentage of the optimal solution (i.e. optimal total value of objects in the bins) that our algorithm generated. Note that in the cases in which the number of bins is greater than one, the bin-packing problem becomes NP-hard, therefore our algorithm only gives an approximation to the actual optimal solution. If the user finds a solution greater than or equal to this approximation, it is considered correct.

2 Technical Documentation

This section gives a brief overview of the Java source code and its relation to the GUI explained in Section 1, as well as the algorithm used to compute the optimal solutions to the knapsack and bin-packing problems.

2.1 Basic dynamics of the Android GUI

The three main views of the application as developed thus far (i.e. the main menu, the high score screen, and the main gameplay screen) are governed by the Java activity classes `AlgovizActivity.java`, `HighScoreActivity.java`, and `BinPackingActivity.java` respectively. These classes provide links between the three views, and contain the main logic for computing the values associated with these views (e.g. the values for the timer, the level numbers, etc.).

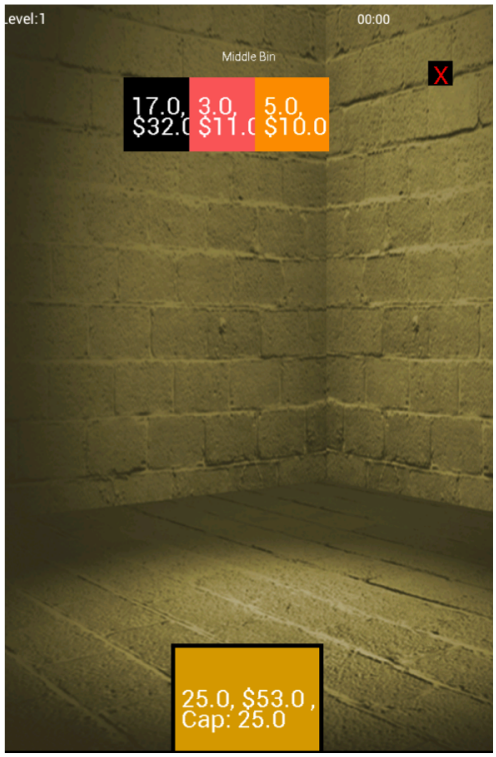

Figure 5: Objects inside bin

The class `BinPackingView.java` includes the vast majority of code governing the graphics of the main gameplay view. Objects are positioned on the screen in this class, and the drag-and-drop capabilities are handled here.

2.2 Supporting classes

The code for the basic infrastructure of the application is contained in the classes `BinObject.java`, `Bin.java`, and `BinObjectPaginator.java`, which all extend `ShapeObject.java`. These classes contain the values associated with each bin and object, the details of how they are to be displayed, etc. `BinObjectPaginator.java` is used to display the objects in a bin, as well as to provide a means of scrolling through objects when there are too many to be viewed on the screen at once.

The `BinPackingProblemFactory.java` class is used to read in problem configurations from a provided `.xml` file and parse the file as to extract the necessary values to be associated with the bins and objects. It also contains the Java implementation of the algorithm used to optimally solve the problems (explained below). The class `Scoreboard.java` is used to manage the mechanics of high score storage.

There are also JUnit test cases written for much of the code, included in the project repository along with the rest of the code.

2.3 Bin-packing algorithm

With only one bin involved, the bin-packing scenario represents an instance of the knapsack problem. To calculate the optimal solution for the one-bin scenario, we thus implemented the dynamic programming knapsack algorithm. We used the algorithm's solution to check against the user's

total "packed-in" value, which allowed us to determine whether the user had reached the problem's true optimal solution.

Scenarios with more than one bin represent instances of the bin-packing problem (which is NP-hard). Here we could have implemented a brute-force approach for calculating the optimal solution. However, since we allow app managers to define an unbounded number of bin objects within problems.xml, the brute-force algorithm's runtime could have quickly escalated to the point of slowing down the app, which could have damaged user experience in the process. Thus, instead of implementing a brute-force approach, we used heuristics to calculate an approximate optimal solution. We based our approach around the knapsack algorithm, which brought our solution's time complexity down to acceptable levels.

To calculate the approximate solution we first made a copy of the set of all bin objects – call this set of objects O'. We then calculated the knapsack solution for a single one of the bins. After calculating the solution, we removed each object that the knapsack algorithm had used from O'. We then repeated the knapsack algorithm for each of the bins, using O' as the input, and removing the set of bin objects that new calculation had used from the set O'. Our approximation approach allows the app to run without any noticeable performance penalties.

Finally, while the approximation algorithm's optimal solution was not necessarily the exact optimal solution, we often found that the figure was high enough to make each level challenging.

2.4 Further information

For further information, please contact the project group members at the email addresses listed on the title page of this document.