Generation of a Four-Way Item-to-Item Navigation Algorithm using Automatic Programming

MORGAN JAKOBSEN and ROLAND OLSSON
Department of Computer Science
Østfold University College
N-1757 Halden
NORWAY

Abstract: With the rising popularity of mobile devices, applications for these devices must meet the challenges of a system with keypad input only. Navigating between randomly placed items using four directional keys can be frustrating if the navigation does not meet the user’s intuitive expectations. Automatic programming with the ADATE system is used to generate a navigation algorithm based on subjective inputs from a user. This means that the user’s mental model of navigation automatically is translated into a program.

Keywords: Program synthesis, automatic programming, mobile devices, graphical user interfaces

1 Introduction

In most computer systems with a graphical user interface, a pointing device such as a mouse is used to select or highlight items on the screen, and to perform actions. Keyboard input is used as a secondary input device, providing handy shortcuts for the experienced user, and of course a method for typing in text. PDAs and similar devices normally provide a pointing stick and a touch-sensitive screen to give the user a method of selecting items. This input method is in many ways similar to the use of a mouse.

However, on simpler devices like cellular phones, a pointing device is often not available. This is reflected in the user interface, which in many cases is based on lists of items that the user can navigate comfortably using the keyboard. With the rising popularity of custom applications and games for these simpler mobile devices, the user interface can no longer be restricted to lists. Using the keyboard as the primary input device presents new challenges, perhaps similar to challenges met in the earlier years of the gaming industry.

In [2], the use of PDAs in working situations is studied using field observations and interviews. One problem revealed is that in many situations, the user’s hands are busy, and that he is not able to operate the PDA without putting his work aside. Ideally, the user should be able to use the PDA as a support tool while performing the primary task at hand. Human-machine interaction with keypad navigation makes it easier to handle the device using only one hand.

One can also argue that most users prefer to use mobile applications with only one hand. A study about the use context and usability problems of the so-called Mobile Internet revealed that most of the time, the users used only one hand when accessing the Mobile Internet [1].

In particular, we have the problem of navigating between a number of items randomly positioned across the screen. This can be done by providing a visible selector, which at all times is located at one of the items. The user can then move the selector from item to item using four buttons representing the four directions up, down, left and right. Some kind of arrow-buttons are provided on all cellular phones, at least they can be represented by the numeric buttons 2, 8, 4 and 6 respectively. For instance, if the user presses the up-button, the selec-
tor skips to an item ”above” the currently selected item.

An example of the situation described can also be seen on the Microsoft Windows desktop\(^1\), where the different shortcuts can be spread randomly across the screen. If a shortcut is highlighted, the directional keys on the keyboard can be used to move the highlighting to other shortcuts.

The user will most likely have an expectation of what should happen when one of the directional buttons is pressed. In many cases, he will not only expect the selector to move in the general direction as instructed, but he has an expectation as to which particular item should be selected. In other words, the user will often have a specific target item in mind when he presses a directional button. To get a high level of usability, it is necessary to try to meet the user’s expectations as to what will happen when he performs a particular action [3]. For this case, it means that the system should try to move the selector to the specific item expected by the user.

There are other important aspects to this topic, including the issue of requiring a consistent behavior from the application, and the fact that the user learns what to expect from the application as he gets more experience using it. A consistent behavior is important, as it allows the user to learn what to expect from the application. However, even though a user knows what to expect, he might not be satisfied with the outcome. The user knows what the system will do in response to his actions, but he would like the system to behave in a different way. In particular, he might be dissatisfied with the order in which items are selected when he uses the directional buttons as described above.

One can argue that there exists an intuitive expectation, shared by most people, as to how the selector should move. For the best usability, the application should then implement this particular algorithm. In figure 1, the problem is illustrated. The items are shown as black dots. The selector, shown as a grey circle, is located at item A. The user has pressed the up-button, as indicated by the arrow. Which item should be selected next? Item B is the item closest to A using Euclidian distance, and it is also the item that has the lowest y-coordinate above than A. Item D is closest to the x-coordinate of A, meaning that selecting item D would lead to the least change in horizontal position, and a movement that follows the general direction indicated by the user, namely upwards. However, the most intuitive selection seems to be item C.

An algorithm must be used to determine the movement of the selector, i.e. what item should be selected next as the user activates the navigational buttons. As described above, the algorithm should meet the user’s expectations regarding this. In this paper, we examine whether automatic programming, and the ADATE system in particular, can be used to generate a suitable algorithm for this four-way item-to-item navigation.

1.1 ADATE

Automatic programming, also known as program induction, is an emerging technology where either a reasonably small program or one or more parts of a potentially huge software system are automatically synthesized. Automatic Design of Algorithms through Evolution (ADATE) [4] is the leading system for induction of functional programs and is likely to eventually be able to generate one or more correct programs for practically any algorithmic problem given enough CPU time.

Under certain assumptions, we have shown [6] that the overall time required for program induction with an old version of ADATE grows as the fourth power of the size of the code to be synthe-

---

1 As observed on Microsoft Windows XP Professional SP 2
sized, which means that ADATE is very computationally demanding.

Therefore, the most immediate practical application is to synthesize rather small programs consisting of less than one hundred lines, say. However, there are lots of problem domains where small programs generated by automatic programming may be superior to any others.

In this paper, we present one example of such a domain, namely generating programs that match the psychological preferences of GUI users by directly optimizing code according to training data collected from the users.

2 Methodology

2.1 Specification of the problem

We define all items as points in a Euclidean 2-dimensional space: \( p_0, p_1, p_2, \ldots \in P \). Point \( p_0 \) represents the currently selected item. The problem can then be seen as picking out the correct point \( p_c \) from \( P \), depending on \( p_0 \) and the direction of movement \( \text{dir} \): \( p_c = f(p_0, \text{dir}) \), \( p_c \neq p_0 \). We transform the coordinates of the points in \( P \) so that \( p_0 \) is at the origin, with coordinates \((0,0)\). Only one of the four directions is taken into consideration, namely the upwards direction. It is assumed that the algorithm is the same for each of the four directions of movement, only rotated 90, 180 or 270 degrees.

2.2 Step-by-step approach

A step-by-step approach is used in the process of solving the given problem, since there is not much knowledge available as to how well ADATE copes with problems of a more mathematical character [5]. The step-by-step approach means that simpler problems with known solutions are solved first, and then the problems are made increasingly difficult until an attempt is made to solve the primary problem. Using this method, it is possible to make necessary adjustments in how ADATE is run, and to verify that the results turn out as expected, before an attempt is made at the primary objective.

An example of a simpler problem is to find the point closest to \( p_0 \). In other words, to find the point which minimizes the distance \( d = \sqrt{x_p * x_p + y_p * y_p} \). This trivial problem is similar in the way that it has the same data-types as input and output, and also that the unknown solution of the main problem might be somewhat similar to this.

2.3 Data generation

A separate tool was developed in order to generate training and evaluation data for the ADATE system. The tool generates scenarios with a set of points according to the specification in section 2.1. The scenarios are presented graphically, and the user must select which of the points he would prefer the selector to move to. The results are stored in a database, and can easily be exported as ADATE input/output data. X values are in the range \([-0.5,0.5]\) and y values in the range \([0.0,1.0]\).

A modified version of the tool was used to generate data for the intermediate test runs, where the problem was defined as picking out the point closest to the origin from a set of points.

2.4 Specification

As shown in [4], an ADATE specification consists of five elements:

1. A set of types.
2. The primitive functions that are to be used in inferred programs.
3. The type of the target function \( f \).
4. A set of sample inputs \( \{I_1, I_2, \ldots, I_{\#1}\} \).
5. An output evaluation function \( \text{oe} \), which uses the set \( \{(I_1, f(I_1)), (I_2, f(I_2)), \ldots, (I_{\#1}, f(I_{\#1}))\} \) to rate the programs.

2.4.1 Set of types

A separate data type representing a point is created. The data type, named "point", consists of two real values, namely the x and y coordinates of the point. In addition, the type for lists of points is defined.
2.4.2 Primitive functions

It is necessary to specify all functions that ADATE is allowed to use. However, specifying a large set of functions means the run-time increases rapidly. For this particular problem, all the basic mathematical operators were allowed, including a square root function and an absolute value function. In addition, a $dist$ function was used, which calculates Euclidian distance from origin to a point:

$$dist(p) = \sqrt{p_x \times p_x + p_y \times p_y}.$$

2.4.3 The $f$ function

The $f$ function is the target of the learning process, in that the contents of the $f$ function will be generated and evolved through this process. Initially, the function is set to throw an exception. The input parameter for the function is a list of points, and the output is one point. Ideally, the function is supposed to pick out one of the input points and return it. However, since the $f$ function is also allowed to create new points, it is not guaranteed that one of the input points is returned; a newly created point can also be returned.

2.4.4 Input and output data

The data sets for the different runs consist of input-output pairs, where the input data is lists of points, and the output data is one single point, which is the same as one of the input points. The tool described in section 2.3 is used to generate the data sets. Data is split into training and validation parts, with approximately two thirds of the data used for training and the rest for validation. The validation data is not used in the training process but only to verify the generated solutions. For the different runs, different number of cases were used, ranging from 45 cases to 200 cases.

2.4.5 The evaluation function

The evaluation function $oe$ determines if the correct point has been selected, as defined by the list of correct outputs. When considering whether two points are in fact the same, the $x$ and $y$ values of the two points are compared to each other. This allows the $f$ function to create and return a new point, consisting of the correct coordinates.

2.5 ADATE runs

As described in section 2.2, a step-by-step approach was chosen, where the following steps were used:

1. The distance problem, where ADATE was allowed to use the $dist$ function.

2. The distance problem, where ADATE was not allowed to use the $dist$ function.

3. The point-to-point navigation problem. ADATE was allowed to use all functions listed.

In the run for step one, a solution was found very fast, within only four minutes. The step two run was done on a cluster with 12 nodes. A correct solution was then found after about 17 hours, which equals about 200 hours of CPU time on a single CPU.

The results of the point-to-point navigation run, that is step three, are discussed in the following section.

3 Results and analysis

This section describes in more detail the ADATE run for solving the point-to-point navigation problem.

3.1 Output files

A number of different files are generated by ADATE. The log file contains meta-information and a transcript of generated solutions that are inserted into the kingdom, i.e. allowed to exist for some time. The trace file contains information regarding the kingdom and the individuals there. In also contains information about time used for calculations etc. The validation file contains information about how well the generated solutions perform on the validation data.

3.2 Examination of a good solution

3.2.1 What is a good solution?

When it comes to the point-to-point navigation problem, it is difficult to determine whether a “correct” solution has been found, as the correct solution is not known, and the input is based on the
subjective preferences of a user group. It is not necessarily so that a solution which perfectly matches the training and validation data is a perfect algorithm, as the results could be caused by overfitting or coincidence. On the other hand, a very good solution might not solve all cases correctly, as the user could have made errors or some bad decisions in picking the output data.

3.2.2 The selected solution

Considering the number of successfully solved training and verification data cases, and the size of the function, the one listed below can be considered the best solution found. This particular solution successfully solves all thirty training examples and also 13 out of 15 validation examples. Considering the small size of the function, overfitting is probably not the case. The solution was found after less than one hour.

3.2.3 Understanding the solution

The inferred function is listed below. Remember that we are moving "upwards", i.e. in the direction of increasing y-values. In order to make the function more readable, variable names have been replaced with more understandable names:

fun f Xs =
  case Xs of
    nil => (case ~0.942312844202 of X1 => point( X1, X1 ) )
    | cons( P1 as point( P1x, P1y ), Xs1 ) =>
      case f( Xs1 ) of P' =>
      case abs( P1x ) < P1y of false => P'
        | true =>
          case dist( P1 ) < dist( P' ) of false => P'
            | true => P1

To better show the underlying algorithm, the function has been converted to use iteration instead of recursion, as shown using pseudo code below. The algorithm has also been extended to match all four directions of movement from an arbitrary position, where current_pos indicates this initial position.

function f(
  list_of_points,
  direction_of_movement,
  current_pos
) : point
begin
  best_p = bad_initial_value
  for each point p in list_of_points do
    vector v = line from current_pos to p
    if angle between v and direction_of_movement < 45 degrees
      and distance( current_pos, p ) < distance( current_pos, best_p )
      then
        best_p = p
  return best_p
end

The algorithm checks each point in turn, and returns one of them. The point returned is the one closest to the initial position (current_pos), selected from all points where the following criteria is met: The angle between the general direction of movement (direction_of_movement) and the direction from the initial position to the candidate point is less than +/- 45 degrees. Figure 2 illustrates this. Points in darker areas are preferred to the points in lighter areas.

4 Discussion

4.1 Results

When it comes to the problem stated in section 1, the results produced by ADATE seem very good. The generated function examined in section 3.2 appears to work well in determining how to navigate between points. However, two of the verification cases are not correctly solved by this function, which indicates that there are better solutions yet to be found.

ADATE was also able to successfully solve the two distance problems. However, for the second problem, that is without a predefined function to
calculate Euclidean distance, overall CPU time was much longer.

### 4.1.1 Alternative configurations

Alternative methods of defining input and output have been considered. One method is to let the target function $f$ evaluate each point individually, and assign a weight to each point. A function external to $f$ would then pick out the point with the highest weight score.

Such a configuration would probably simplify the $f$ function, as it does not need to enumerate the points, only evaluate them individually. However, the solution space is made much smaller, primarily because the $f$ function is not able to see the points in relation to each other.

### 4.2 Future work

For the problem stated in this paper, there is still room for improvement. As mentioned in section 4.1, there were two validation cases that did not match the generated function examined in section 3.2. This indicates that even better solutions can be found. To examine this, more training data is probably necessary, and perhaps the data should be more carefully constructed, and not only contain randomly generated instances. For example, instances could be generated that lie along the borders of the generated solution, so that ADATE is encouraged to examine the border cases more closely.

### References


