Pareto Optimal User Interface Design

Sujay Jayakar

***Abstract*—User interface design is more an art than a science, yet developers do not have the luxury of designing unique interfaces for every user's needs. Therefore, in situations where software designers cannot accommodate the entirety of user diversity, optimization based approaches can fill in the gaps. We present a method of designing interfaces using multi-objective optimization. When performing our static optimization computation, we defer forcing trade offs between different objectives, yielding a set of Pareto optimal solutions. Once the user has declared his or her preferences, a secondary ordering on the Pareto set can then find a unique optimal interface for that individual. Since we perform most of the optimization beforehand, we can incorporate global characteristics that real time implementations struggle to capture. The benefit of our approach is largely practical. The computation we perform, the optimization with respect to multiple objectives, could be performed similarly in theory by constraining all of the objectives save one and using the current approaches to single-objective based user interface optimization. This alternative approach, however, is unfeasibly expensive in the face of many, opposing objectives, and exploiting the multi-objective nature of the problem cuts down on the time needed substantially. Furthermore, our approach intelligently distributes the computational burden of optimization between design time and run time, which stands in sharp contrast with the monolithic run time only and design time only approaches.**

I. Introduction

COMPUTER user interfaces are typically designed by hand by software designers, who aim to maximize the utility of their software to their user base. Artificial intelligence researchers have long proposed to leave this optimization to the computer. This suggestion has been met with skepticism from the HCI community. We acknowledge the difficulty of procuring acceptable machine designed interfaces, but there remain situations where such automatically generated interfaces are appropriate. Interface designers typically target a single class of user and platform, with no variance allowed. We propose using the technique of multi-objective optimization to introduce a new level of flexibility in a program's interface. The quality of interfaces is governed by a plethora of objectives, which are often orthogonal and hard to compare. Traditional approaches force a trade off between these objectives during the optimization phase, which may lead to a solution that does not suit the user's preferences. Trade offs may take the form of some linear combination of the different objectives with varying weights: The problem remains of intelligently choosing the weights, and even with all possible weights, single-objective optimization may not identify all optimal solutions. Our solution, on the other hand, defers this trade-off to the end user, whose choices can single out a particular solution for his or her needs.

II. Related Research

Multi-objective optimization is a rich field of research with many novel approaches. The first algorithm we will employ is a multi-objective genetic algorithm entitled *NSGA-II* [1]. Given additional time, we will explore the effects of using alternate algorithms, such as evolutionary covariance matrix adaptation and the Metropolis algorithm [2] [3]. We will use the ECsPy toolkit for multi-objective optimization as a common basis for evaluating these different solvers.

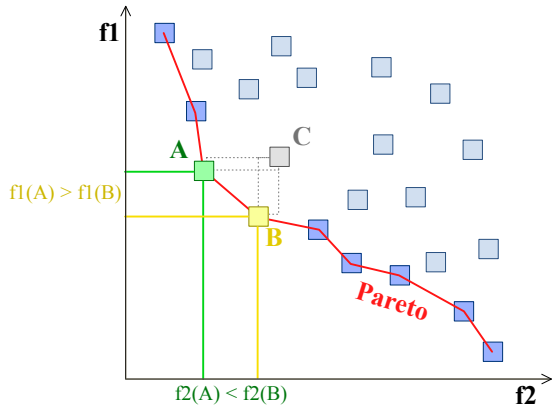

Fig. 1. An illustration of multi-objective optimization with respect to two objectives, f_1 and f_2. The boxes represent feasible solutions, and the darkened solutions form the Pareto optimal set. Note that moving from A to B improves f_2 at the cost of f_1. Source: Wikipedia

On a more general level, the idea of creating machine-designed interfaces through optimization has an extensive research history. Automated design has been attempted since the 1980s with projects such as Cousin, Mickey, and Humanoid [4] [5] [6]. Zhai designed a keyboard using a sophisticated physically based method derived from the Metropolis algorithm [7]. Hinkle employed genetic algorithms to design keyboards for Brahmic scripts, a difficult problem that involved the allocation of over 3,000 character combinations [8]. Gajos and Weld took a different approach with their Supple software, which designs interfaces in real-time with respect to a user's history [9].

III. Problem Description

In this paper, we reduce the task of user interface design to multi-objective optimization. Formally, we have a set of feasible interface designs $\mathcal{S}$ and a sequence of objective functions $f_i : \mathcal{S} \mapsto \mathcal{R}_+$ which we would like to optimize. Using the f_i, we induce a partial order $\succ$ on $\mathcal{S}$ by the typical Pareto criterion. A solution a *Pareto dominates* another solution b if a is at least as good as b with respect to each objective function f_i, and a is strictly better than b for at least one of them. It is straightforward to show that $\succ$ is, in fact, a partial order. From this partial order, we say a solution s is *Pareto optimal* if it is not Pareto dominated by any other solution. Our goal is to find the set of all Pareto optimal

solutions $\mathcal{S}^*$, which we will designate as the *Pareto front*. Intuitively, there are no more "free lunches" past the Pareto front: Improving one of the objectives requires worsening at least one of the others.

Once the Pareto optimal set has been identified, the user can supply his or her own ordering $\succ_U$ on the Pareto front, identifying a single optimal solution. The user's preference $\succ_U$ encodes the trade offs the user is willing to make between the different objectives. Since most of the optimization has been done statically ahead of time, identifying the optimal solution with respect to $\succ_U$ is trivial.

For this project, we chose to use evolutionary algorithms to generate the Pareto optimal set. For our results on multi-objective optimization on Indic language keyboards, the set generated was robust to restrictions on individual objectives, meeting our goals of adaptability to user needs. However, our task in general of using multi-objective optimization to design interfaces does not require the use of genetic algorithms, and fairly simple dynamic programming approaches to multi-objective optimization exist [10]. Since our problem representation lends itself well to evolutionary computation, we focused on only the genetic algorithmic approach.

We choose to represent solutions as mappings from abstract interface specifications to concrete widget representations. The core of an abstract interface lies in *primitive types*, denoted by τ, which request integers, strings, floats, and Booleans from users. A designer can add a constraint $\mathcal{C}_\tau$ over the domain of a primitive type to form a *constrained type* $\langle \tau, \mathcal{C}_\tau \rangle$. Types can be composed into *container types* $\{\tau_1, \tau_2, \ldots, \tau_n\}$ that represent some semantic grouping of interface elements.

A potential solution maps an abstract interface representation to a concrete description in terms of widgets. Each platform is a set of GUI widgets that are available to the programmer. An interface description specifies the implementation of each abstract element in terms of widget choice and parameters. Given different objectives, solutions will intelligently choose widgets, preserve container groupings, and tune widget parameters.

IV. Implementation

A. Abstract UI

The user interface design process here begins with the abstract representation of the user interface, provided by the application designer. We provide a simple XML representation along with a parser that creates the Python abstract tree data structure. The allowed container types are `GenContainer` and `IdContainer`. The `GenContainer` type does not enforce any structure on the relationships between its children elements. The `IdContainer` type, on the other hand, imposes the constraint that all of its children must have the same presentation. Both container types take an `ordered` argument as well as a `name`.

The primitive types currently supported are `AbstrInt` for integers, `AbstrFlt` for floating point numbers, `AbstStr` for strings, and `AbstrBool` for Booleans. The abstract integer type takes a `loRange` argument and `hiRange` argument, indicating the lower and upper bounds for the input. If they are not specified, they default to `0` and `10`, respectively. The abstract float type takes the same arguments and defaults to the same values.

```
<?xml version="1.0"?>
<GenContainer name="Classroom Management">
  <IdContainer name="Light Bank" ordered="True">
    <GenContainer name="Left">
      <AbstrBool name="Light"></AbstrBool>
      <AbstrFlt name="Level"></AbstrFlt>
    </GenContainer>
    <GenContainer name="Center">
      <AbstrBool name="Light"> </AbstrBool>
      <AbstrFlt name="Level"></AbstrFlt>
    </GenContainer>
    <GenContainer name="Right">
      <AbstrBool name="Light"></AbstrBool>
      <AbstrFlt name="Level"></AbstrFlt>
    </GenContainer>
  </IdContainer>
  <GenContainer name="A/V Controls">
    <GenContainer name="Projector">
      <AbstrBool name="Power"></AbstrBool>
      <AbstrStr name="Input"
       allowed="Computer 1;Computer 2;Video">
      </AbstrStr>
    </GenContainer>
    <AbstrBool name="Screen"></AbstrBool>
  </GenContainer>
  <AbstrStr name="Vent"
   allowed="Off;Low;Med;High"></AbstrStr>
</GenContainer>
```

Fig. 3. The XML representation of the classroom user interface outlined in Figure 2. Note that the description is entirely abstract: There are no references to size or position other than the semantic groupings provided by the container types.

The abstract string type serves two purposes. First, if the `allowed` parameter is left empty, the program requests a string from the user in the form of a text input. Second, the developer can indicate a list of allowed choices in the form of a semicolon separated string. Therefore, if the `allowed` parameter is nonempty, the program will present the user with a choice between different strings. The abstract Boolean type does not take any arguments other than its name.

If the developer wishes, he or she can specify a description of the element for internal use by providing a documentation string between the opening and closing tags for a particular element. These do not appear in the final implementation, but may be useful for debugging and development.

Although this approach is limited to static user interfaces, such as control panels and input interfaces, the hierarchical approach derived from combining nested containers with basic action-driven types allows for a great deal of expressive power. Given the flexibility of our abstract framework and the Kivy library[1], adding functionality for dynamic interfaces would not prove to be difficult.

B. Constraints

Fundamental to any genetic algorithm chromosome representation is a set of *constraints* on the legal values of the

[1]http://www.kivy.org

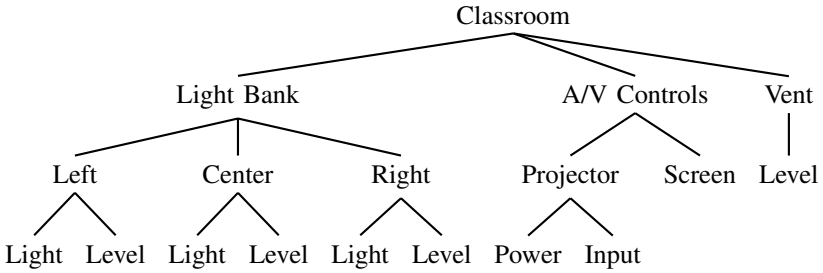

Fig. 2. The abstract representation of the classroom user interface detailed in SUPPLE. Nodes of the tree are container types, represented as sequences of their children. Container types, such as "Light Bank" admit constraints. For example, the presentation of "Left," "Center," and "Right," must all be identical. Leaves of the tree represent primitive types.

alleles. Our set of constraints is especially complicated due to the tree based nature of our problem. Each abstract element can have a set of allowed widgets, and each widget has a set of constraints on its presentation. Therefore, the number of alleles in a given chromosome is not constant, as mutating one widget to another can change the number of alleles. Therefore, we choose to abstract away the details of these constraints to a generic constraint class. This class provides the functionality for storing a constraint's allowed values as well as randomly generating a value for the chromosome.

The constraint class facilitates the choice between both different widget elements and different values for those widgets. Since we have wrapped up the myriad individual constraints within a constraint class, the length of a chromosome remains the same, making the genetic operators much more manageable.

C. Constraint Trees and Arrays

To make the interface between the abstract representation and the chromosomes dictating a particular UI presentation, we need to interpret the abstract tree as a set of constraints on the UI search space. We accomplish this transformation using a collection of rewrite rules on the abstract tree.

We begin by specifying the set of allowed interface widgets and their properties. These choices are largely dictated by KIVY, the user interface library we decided to use for this project. The first is a slider, which has three constraints: orientation, lower bound, and higher bound. The second is a switch, which does not have any constraints. The third is a button, which takes a *group* name as a constraint. Buttons within the same group are presented together, and, much like a radio button, choosing one button disengages any other buttons in its group. The toggle element has a similar group constraint. The final primitive widget is the text input field, which has a constraint indicating whether multi-line input is allowed.

We currently have two layout widgets implemented. The first is the `BoxLayout`, which organizes widgets in a linear fashion. The choice between a vertical or horizontal presentation is encoded as a Boolean constraint. Each child element has a floating point constraint associated that represents the size hint passed to KIVY.

The `GridLayout` layout widget arranges its children widgets in a grid pattern. For a container of n elements, the grid constraint chooses the number of columns c and rows r such that $r \times c = n$. These factors are passed along as the allowed values for the column and row parameters. Currently, we have chosen to only allow multiples, which may lead to strange results if, for example, the number of children widgets is prime. In this case, the designer should add blank padding elements. At the moment, all columns and rows are required to be the same size, which is obtained by dividing the widget's space evenly.

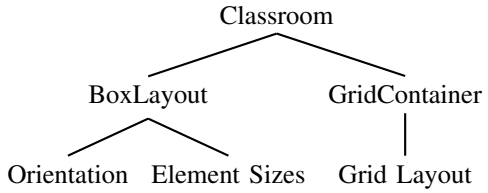

Fig. 4. The UI widgets and constraints associated with a single abstract element. Each element of the constraint tree is paired with sub-tree like this one. Optimizing over trees, much less nested trees, is not straightforward with a genetic algorithm, so we pack the constraint tree into a corresponding constraint array.

The abstract tree derived from the XML parse is then traversed using the following rewrite rules.

$$
\begin{aligned}
GenContainer &\rightarrow \{BoxLayout, GridContainer\} \\
IdContainer &\rightarrow \{BoxLayout, GridContainer\} \\
AbstrInt &\rightarrow \{Slider, TextInput\} \\
AbstrFlt &\rightarrow \{Slider, TextInput\} \\
AbstrStr &\rightarrow \{TextInput\} \\
AbstrBool &\rightarrow \{Toggle, Switch\}
\end{aligned}
$$

After the rewriting process has completed, each element has a set of associated widgets. Each widget, in turn, has a set of constraints, forming a tree-like structure per abstract element. As noted before, we need to turn this tree into a genetic algorithm friendly array data structure. This process is completed by the `makeArray` method which traverses the tree in preorder, keeping a parent pointer for each element to facilitate tree construction. Since the structure of the tree is preserved under all transformations, the preorder traversal is identical for all chromosomes. This invariant allows us to splice chromosomes together without fear of generating a

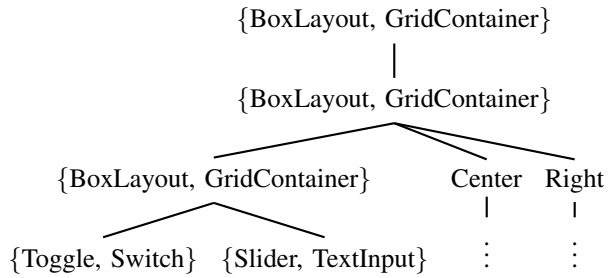

Fig. 5. A portion of the constraint tree created after rewriting the abstract tree from the XML parse. Each element has a set of associated constraints, as detailed in Figure 4.

malformed tree. This constraint array produced in the tree unwinding is kept throughout the entire algorithm, as generating new nodes under, for example, the mutation operator requires the enumeration of allowed values provided by the constraint array.

D. Genetic Operators and the Algorithm

Once we have this constraint array in hand, we can generate chromosomes randomly using our generator operator, which iterates over the constraints and produces satisfying values. This functionality is implemented by a function class, which internally stores the constraint array as a template for the chromosomes upon initialization. This object is then passed to the ECSPY library[2], which generates the initial population.

The second operator is the combinator, which takes two chromosomes and returns two children chromosomes that are the product of combining its parents. Because of our previous design choices, the implementation of this operator is no more difficult than typical array-based evolutionary operators. Our implementation is a simple implementation of single-point crossover that randomly chooses a crossover point within the chromosome and then splices the two parents in both combinations to produce the two offspring, which are returned to the genetic algorithm.

The final operator is the mutator, which takes a chromosome and alters it with some fixed probability μ, which is set by the user at the beginning of the computation. The mutator operator traverses the chromosome array, marking a node for alteration with probability μ. The nodes marked for alteration are then replaced with a new node instances from their corresponding constraint array node. This approach helps the algorithm explore the search space while ensuring mutated solutions are still legal user interfaces.

The final component remaining for the genetic algorithm is the set of fitness functions, which are the heart of our multi-objective approach. We implemented two fitness functions as examples, but any function can be used, so long as it takes a chromosome and returns a floating point number. The first is the size evaluator, which, roughly, returns the size remaining in the device after the interface has been rendered. More precisely, the evaluator requires that the interface code be in the following format.

[2]http://code.google.com/p/ecspy

```
<GenContainer name = "Device Wrapper">
  <GenContainer name = "Whitespace">
    </GenContainer>
  <GenContainer name = "UI">
    [UI description    ]
    [specified by user ]
  </GenContainer>
</GenContainer>
```

The function then returns the size of the "Whitespace" container, which is to be maximized. This function heavily depends on the size tree implementation in the rendering module, which will be described later. The nature of this objective fits closely with the goal of designing keyboards for mobile devices, which live in a fixed width region on the bottom of a phone's screen. Maximizing the size available to the device for other purposes marks an optimal solution.

The second evaluator is based on Fitts' law, which gives a theoretical estimate of the time needed to navigate the interface [11]. More specifically, it gives us the mean time needed to type two characters using some form of pointing device. The distance between each key is divided by the width of the target key and then added to one. The logarithm, base two, of this result is then weighted by the frequency of this pair of characters and then divided by the "Index of Performance" (IP), which we, in order to maintain consistency with previous research, set at 4.9. Therefore, the mean time to type a character is

$$\bar{t} = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{P_{ij}}{IP} \left[\log_2 \left(\frac{D_{ij}}{W_j} + 1 \right) \right].$$

This computation first requires a dictionary of the pairwise distance of the characters. To provide this data, we interface the evaluator again with the rendering module of our library. For each chromosome passed to the evaluator, the function rebuilds the UI tree from the chromosome, traverses the tree, computing the size of each element, and then builds a dictionary mapping element names to their sizes and positions. This dictionary is also used to compute the width data for each element.

The pairwise frequencies are read in from a text file provided by the user. In our implementation for the Assamese language, manipulating Unicode characters proved to be difficult, so we mapped each Unicode character to a unique integer. These integers are then passed around the optimization routine in place of the actual Assamese characters and then substituted in final rendering process.

We used the ECSPY library's implementation of the NSGA II algorithm. The library was flexible enough to accommodate our mutator, combinator, and generator classes described above. Each class was instantiated with the relevant constraint array and then passed to the optimization routine.

Specifically, we used our own mutators, generators, and combinators along with the `generation_termination` terminator and the `file_observer` observer, which intermittently dumped statistics about the evolutionary algorithm as

Fig. 6. A simple, abstracted example of a keyboard interface designed in the KIVY UI framework.

well as a list of the individuals in the population to a file. Since we ensured that our operators did not generate any illegitimate solutions, the `nsga2.bounder` was not needed and was set to an identity function.

E. UI Implementation

The KIVY framework for Python provides an excellent cross platform framework for implementing the results from the genetic algorithm. The library's simplicity makes it perfect for our algorithm, which focuses on static interfaces. Another benefit is that KIVY supports both desktop and Android applications, allowing us to perform actual tests on devices with especially relevant size constraints.

There is a direct correspondence between the widgets chosen in the rewrite rules described previously and the widgets available in the KIVY library. From the side of understanding the UI library, we have designed simple demonstrations in an abstract manner that will hopefully interface well with our chromosome specification.

The process of generating a KIVY interface from a chromosome generated by the evolutionary algorithm is involved. As noted before in our discussion on our evaluation functions, the first task is to generate a ***size tree*** which maps elements to their position and size.

This task is accomplished by first rebuilding the chromosome tree from the linear array specified by the chromosome. This is largely a matter of traversing the array and reconnecting the parent pointers generated from the tree unwinding during the formation of the constraint array. We then proceed from the root node, taking the total interface size from the problem configuration. If the current node is a `BoxLayout` element, the children sizes are determined by their weights w_i.

$$\mathsf{width}_i = \frac{\mathsf{width}_{\mathsf{parent}} * w_i}{\sum w_i}$$
$$\mathsf{height}_i = \frac{\mathsf{height}_{\mathsf{parent}} * w_i}{\sum w_i}$$

Similarly, the permutation information is read from the chromosome. The permutation constraint for a container of n elements is a permutation of the sequence $\{0, 1, \ldots, n-1\}$ that maps the array index of the child to its position. One relevant concern with this approach is that the number of permutations for a container of n children grows as $n!$, which may exceed the period of the random number generator used. If this is the case, the entire solution space may not be explored, which may adversely affect problems sensitive to element permutation (such as keyboards). Fortunately, the Mersenne twister generator used in our implementation has a period that well exceeds $62!$, where 62 is the number of keys in our example.

The case for `GridContainers` is similar. We read in the number of columns and rows from the generated list of factors encoded in the constraint array. For now, the heights and widths of the rows and columns are assumed to be uniform, so determining the sizes and positions of the children is not difficult.

$$\mathsf{width}_i = \frac{\mathsf{width}_{\mathsf{parent}}}{n_{\mathsf{cols}}}$$
$$\mathsf{height}_i = \frac{\mathsf{height}_{\mathsf{parent}}}{n_{\mathsf{cols}}}$$

Finally, if an element is a base type, it simply inherits its size from its immediate parent. Now that the sizes and positions have been determined, we are finally in a position to build the KIVY representation of the interface. The `buildWidgets` method takes the sizes from the size tree previously built and sets the appropriate `size` attributes for the newly generated widget. The method then recursively calls itself on all of the element's children to form the widget hierarchy, which is then passed to KIVY to create an application.

V. RESULTS

We focused our multi-objective approach on the task of designing optimal keyboards for Indic languages to fit in with previous research. More specifically, we aimed to design keyboards for platforms on which space is at a premium. Therefore, our two objectives were speed of use, as determined by Fitts' law, and keyboard size.

As such, we coded an optimized version of our multi-objective algorithm to focus on keyboards alone. We first designed an optimized chromosome that is just an array of integers and floats. The first sixty-two elements are integers that represent the permutation of the keys, as with our abstract approach from before. The next number is a float that represents the normalized height of the device and ranges from zero to one.

A. Keyboard Implementation

Generation of these chromosomes is straightforward, as we simply create an array of integers ranging from 0 to $n-1$, where n is the number of characters in the desired language, and then shuffle the array. Next, we append a random number from zero to one to represent the height of the user interface.

Mutation is similar. With probability μ, we shuffle the first n elements of the array using the built in library function `random.shuffle`. Finally, we choose another random floating point number between zero and one for the height parameter.

We use ECSPY's built in differential crossover combinator to create children chromosomes from parent chromosomes. Since the crossover method does not guarantee that the results will be legitimate permutations of $\{0, 1, \ldots, n-1\}$, we wrote

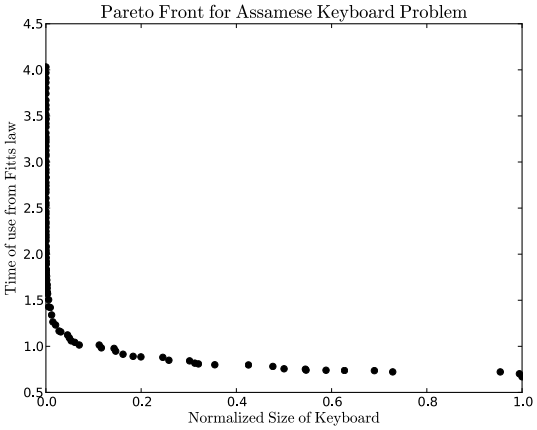

Fig. 7. The Pareto optimal front for an Assamese keyboard design problem on a mobile device. Note that decreasing the mean time per keystroke requires increasing the size of the keyboard, indicating that the two objectives are opposing, as conjectured earlier. Therefore, imposing a maximum size on the keyboard implicitly bounds the maximum efficiency of the keyboard. Now that we have this set in hand, we can, at run time, determine the size constraints for the user and then choose the best solution that satisfies these constraints and maximizes the other objective.

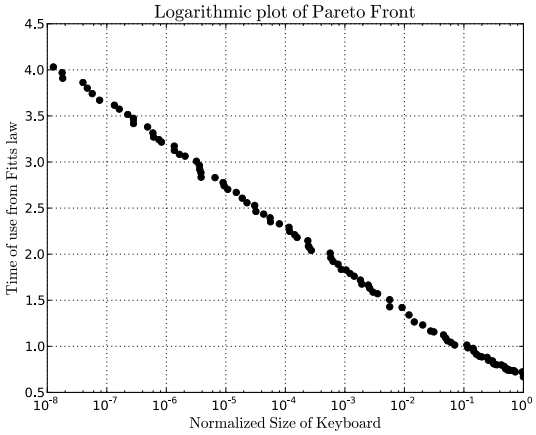

Fig. 8. A logarithmic plot of the Pareto frontier from the previous figure. The trade off between size and efficiency only becomes relevant with smaller sizes, so expanding the horizontal axis renders the plot much more readable.

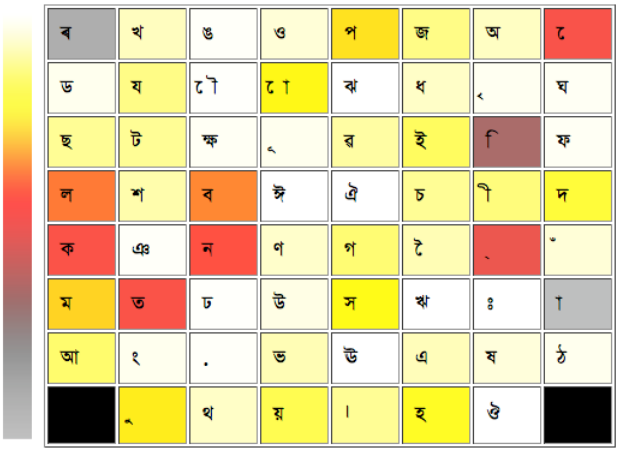

Fig. 9. The keyboard generated when the normalized size constraint of 0.3 was imposed. This keyboard has a fitness of 0.848804 according to Fitts' law, which corresponds to 14.14 words per minute.

a *bounder* function that takes an illegitimate chromosome and restores our preferred invariants.

The bounding function first considers the first $n-1$ elements that correspond to the key permutation. It identifies which keys are missing and which keys are repeated. It then randomly assigns missing keys to repeat positions until all keys are present. Finally, it ensures that the final element, the floating point number representing the height of the keyboard, is within the interval $(0, 1]$.

The evaluators used are identical to the Fitts' law evaluator and size evaluation described in the previous section on abstract interfaces. The pairwise frequencies were read in from the Assamese corpus generated from WIKIPEDIA articles.

The Assamese language has 62 characters, all of which must be accommodated on the onscreen keyboard. Each character is associated with a pop up box of conjoints, which were designed by hand and not by the algorithm. This approach was taken from the work by Hinkle, who used a single objective genetic algorithm in a similar spirit [8].

B. Data

We present the Pareto fronts generated by our algorithm running on the Assamese keyboard. There is a clear trade off between size and efficiency, especially once the size becomes especially small. The generation of this set took roughly 12 hours of computation, but once it has been generated, the process of identifying the optimal solution with respect to a particular constraint is instantaneous.

The Pareto frontier was unexpectedly well behaved. Given the complex nature of our optimization problem, we did not expect such a well behaved convex frontier. Furthermore, when viewed on a logarithmic plot, the frontier is linear, which may be related to the $\log$ term in Fitts' law and the linear dependence on the height of the chromosome. Although we have no proof of convexity, it would be interesting to explore the results of using convex optimization routines, which are much quicker and more robust than our evolutionary algorithms, on this specific optimization problem.

To visualize the results of the keyboard optimization, we developed a special renderer for this problem. We present the keyboard with each key colored according to a heat map that represents the frequency of the character encoded by that key. The heat map legend is presented in the first keyboard. The frequencies of the characters were read in from corpus data and then normalized to one.

After relaxing the size constraint past 0.5, increasing the allowed size of the device did not improve its efficiency, and the optimal solution satisfying the constraint remained the same.

C. Gujarati Keyboard

In addition to running our algorithm on the Assamese language, we developed keyboards for the Gujarati language as well. The Gujarati language has 64 base characters, while

Fig. 10. The keyboard generated when the normalized size constraint of 0.4 was imposed. This keyboard has a fitness of 0.799940 according to Fitts' law, which corresponds to 15.00 words per minute.

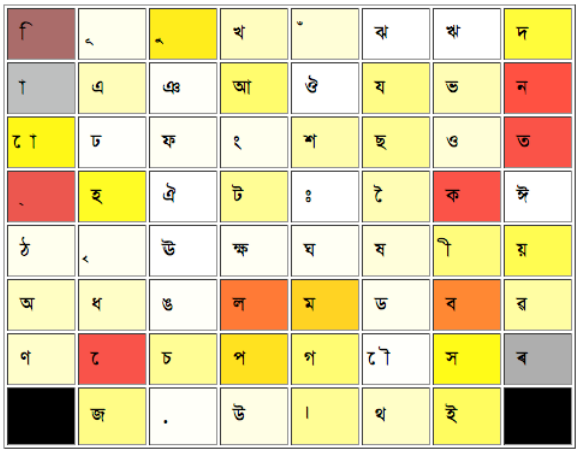

Fig. 11. The keyboard generated when the normalized size constraint of 0.5 was imposed. This keyboard has a fitness of 0.781674 according to Fitts' law, which corresponds to 15.35 words per minute.

Assamese has 62. Therefore, the same 8×8 configuration worked, although the Gujarati keyboard did not have the blank spaces in the corners that the Assamese keyboard did. The mean time per keystroke derived from Fitts' law for the Gujarati keyboards was higher than the solutions for the Assamese keyboards.

We produced similar plots for the Pareto front for the Gujarati keyboard problem. The Pareto front is fairly similar to the Assamese front in that it is convex and exponential. Since the optimization problems use the same chromosome encoding and fitness functions, the similarity is unsurprising.

As with the Assamese problem, relaxing the size constraint past 0.5 did not significantly increase the efficiency of the keyboards. We present three keyboards here in the Gujarati language using the same size constraints from the previous problem.

VI. Conclusions

Casting user interface design as a numerical optimization problem is hardly a new approach, but increasing the richness

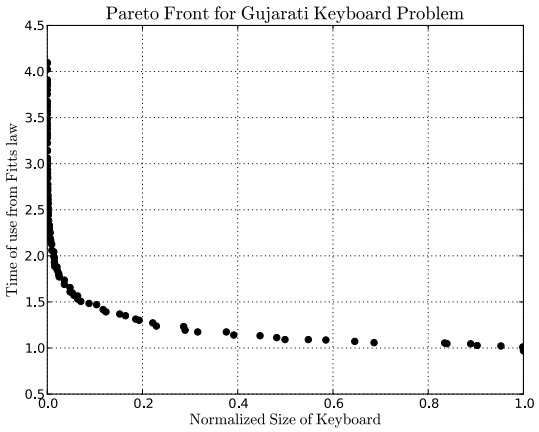

Fig. 12. The Pareto optimal front for the Gujarati keyboard design problem on a mobile device. The front is monotone and convex, just as the Pareto front for the Assamese keyboard design problem.

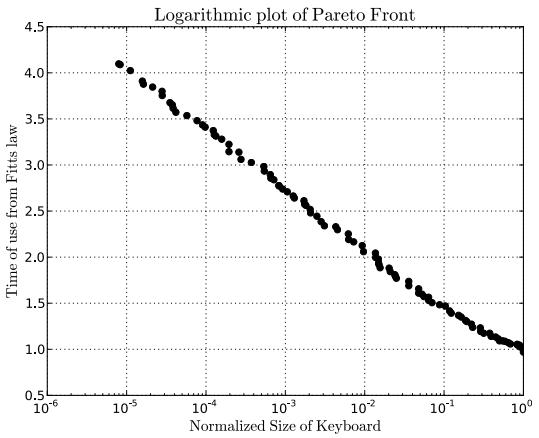

Fig. 13. As before, plotting the Pareto front on a semi-logarithmic plot reveals its simple exponential figure.

Fig. 14. The Gujarati keyboard generated under the size constraint of 0.3. This keyboard has a mean time per keystroke of 1.193138 seconds, which corresponds to 10.06 words per minute.

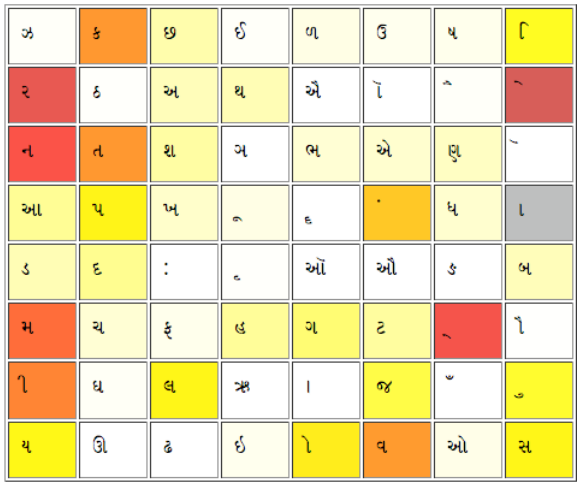

Fig. 15. The Gujarati keyboard generated under the size constraint of 0.4. This keyboard has a mean time per keystroke of 1.140539 seconds, which corresponds to 10.52 words per minute.

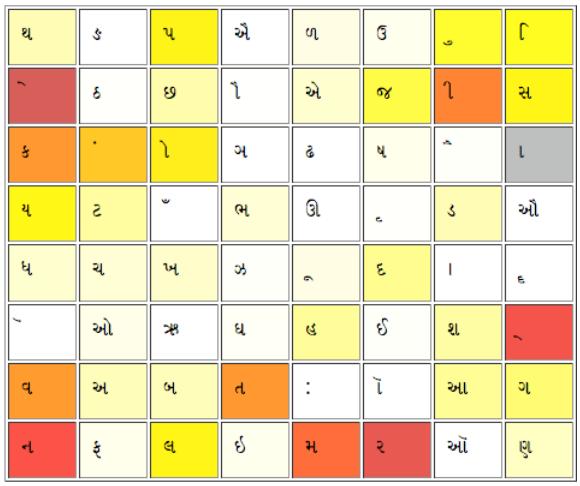

Fig. 16. The Gujarati keyboard generated under the size constraint of 0.5. This keyboard has a mean time per keystroke of 1.091797 seconds, which corresponds to 10.99 words per minute.

of the optimization by simultaneously considering multiple objectives has produced offline algorithms that, in spite of their offline nature, can still be responsive to users' needs. This approach yields a good compromise between entirely static methods that entirely determine the interface ahead of time and fully real-time approaches whose objectives must conform to their heuristics.

Our results with keyboard design confirmed our hypothesis: Multi-objective evolutionary algorithms can identify a Pareto front that represents trade offs between the opposing objectives. Constraining all but one of the objectives then provides an efficient way to optimize relative to the users' needs while benefiting from the flexibility of static offline optimization algorithms.

Acknowledgments

The research reported in this document has been funded partially by NSF grants CNS-0958576 and CNS-0851783.

References

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, April 2002.

[2] C. Igel, N. Hansen, and S. Roth, "Covariance Matrix Adaptation for Multi-objective Optimization," *Evolutionary Computation*, 2007.

[3] J. Vrugt, H. Gupta, W. Bouten, and S. Sorooshian, "A Shuffled Complex Evolution Metropolis Algorithm for Optimization and Uncertainty Assessment of Hydrologic Model Parameters," 2003.

[4] P. J. Hayes, P. A. Szekely, and R. A. Lerner, "Design Alternatives for User Interface Management Systems Based on Experience with Cousin," in *CHI '85: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, 1985, pp. 169 – 175.

[5] D. R. Olsen, "A Programming Language Basis for User Interface," in *CHI '89: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, 1989, pp. 171 – 176.

[6] P. A. Szekely, P. Luo, and R. Reches, "Facilitating the Exploration of Interface Design Alternatives: The Humanoid Model of Interface Design," in *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, 1992, pp. 507 – 515.

[7] S. Zhai, M. Hunter, and B. Smith, "The Metropolis Keyboard–An Exploration of Quantitative Techniques for Virtual Keyboard Design," 2000.

[8] L. Hinkle, M. Lezcano, and J. Kalita, "Designing Soft Keyboards for Brahmic Scripts," in *International Conference on Natural Language Processing*, Kharagpur, India, 2010, pp. 191–200.

[9] K. Gajos, D. Weld, and J. Wobbrock, "Automatically Generating Personalized User Interfaces with SUPPLE," *Artificial Intelligence*, 2010.

[10] R. Kltzler, "Multiobjective dynamic programming," *Optimization*, vol. 9, pp. 423–426, 1978.

[11] P. Fitts, "The information capacity of the human motor system in controlling the amplitude of movement," *Journal of Experimental Psychologyu*, 1954.