Visualizing Optimization Algorithms via Rapid Prototyping of Graphical User Interfaces

Dr. John F. Beetem
The MITRE Corporation
11493 Sunset Hills Road
Reston, VA 22090, USA
beetem@mitre.org

June 8, 1992

Abstract

Developing new optimization algorithms is often a process of trial and error, especially in multivariate situations. Graphical visualization of algorithm behavior can be a powerful technique for assessing the effectiveness of different algorithms and heuristics, provided that the Graphical User Interface (GUI) can be prototyped and modified quickly. This paper describes how visualization was used to prototype a placement and routing package for the MITRE Digital Transform Machine (DTM), a powerful reconfigurable logic array. Using the Galaxy Programming Language and Environment greatly simplified GUI construction, allowing the prototyper to concentrate on placement and routing algorithms instead of the details of conventional GUI programming.

Introduction

Exploring new algorithms is a process of trial and error, especially in complex, multivariate situations. This is particularly true for optimization algorithms, which often require heuristics to produce good solutions within acceptable processing time. Quickly devising and testing many different heuristics requires rapid software prototyping for implementation of the heuristics and effective visualization methods for assessment.

To develop optimization algorithms efficiently, the underlying software engineering environment should take care of as many details as possible so that the developer's mind can concentrate on creative problem solving. One should be able to modify algorithms or heuristics and see the results instantly, preferably in a highly visual form [3].

Unfortunately, programming a high-quality Graphical User Interface (GUI) is very time-consuming with conventional languages and window managers. Even something as conceptually simple as creating a window containing “Hello, world!” can take pages of code and expert programmers. To use graphics for visualizing algorithms, prototyping GUI code has to be much simpler.

This paper describes how visualization was used to prototype a placement and routing package for the MITRE Digital Transform Machine (DTM), a powerful reconfigurable logic array. Using the Galaxy Programming Language and Environment greatly simplified GUI
fn RefreshPart = draw "Hello, world" at ([10,10]);

fn main =
( assert (create window Next Rect <> Nil):
"Could not create window";
SelWin.RefreshPart := fnadx RefreshPart;
window title := "Hello!"
)

Figure 1: Simple “Hello, world” program.

coloration, allowing the prototyper to concentrate on effective visualization and placement and routing algorithms instead of being bogged down in the details of conventional GUI programming.

The Galaxy Programming Language and Environment

Galaxy [1, 2] is a new programming environment intended to improve programmer productivity by at least an order of magnitude. Galaxy is a multi-window environment supporting simultaneous editing, compiling, and testing. Galaxy’s incremental compiler minimizes the time needed to recompile a program modification, encouraging experimentation. Algorithms can be modified and evaluated literally in seconds.

Galaxy endeavors to make window management as easy as possible by providing a simple, portable window manager for prototyping GUIs. The Galaxy Simplified Window Manager (G-SWIM) provides an abstract layer between Galaxy programs and the underlying operating system and window manager, e.g., X Windows or Macintosh operating system. G-SWIM hides the complexity of the underlying manager by converting high-level operations such as “create window” and “draw line from zi to z2” into the corresponding machine-specific calls. Often a single G-SWIM operation corresponds to dozens of lines of X Windows or Macintosh code.

Galaxy application programs only have G-SWIM calls: they never call the operating system directly. This makes Galaxy programs completely portable between hardware platforms. Between 680x0 platforms, Galaxy programs do not even require recompilation.

G-SWIM and Galaxy applications work together as partners. G-SWIM handles generic window functions, such as scheduling redrawing of exposed windows, clipping, scrolling, and when to blink cursors. The application program specifies how to refresh all or part of a window, and how to respond to user interactions like keystrokes, mouse movement, and mouse buttons.

Fig. 1 shows a simple Galaxy program for creating a window and displaying “Hello, world” in it. Running this program produces the screen shown in Fig. 2. Two functions are defined. Function RefreshPart specifies how to redraw the contents of the window.

---

1Galaxy currently runs on Apple Macintosh, HP9000/3XX/4XX, Sun-3, and Atari ST computers.
fn RefreshPart = draw "Hello, world" at ([10,10]);

fn main =
(assert (create window Next Rect <> Nil):
  "Could not create window";
SelWin.RefreshPart:= fnadx RefreshPart;
window title:= "Hello!";
window limits:= ([0,0],[500,300])
)

Figure 3: Scrollable "Hello, world" program.

G-SWIM calls this function whenever the window becomes exposed, e.g., if a window in front of it is deleted.

Function main first creates the window, checking this was done successfully. It then sets RefreshPart to be the function called when the window needs refreshing. Finally, main sets the window title to "Hello!".

Running main creates a "Hello!" window. After main terminates, the "Hello!" window continues to exist as an object, redrawing itself as needed, until it is closed by the user explicitly.

Adding scrolling to the "Hello, world" program requires only one line of code, as shown in Fig. 3. The window limits expression defines the maximum values of user coordinates that can be displayed in the window. Running the modified program produces Fig. 4, where scroll bars are drawn automatically by G-SWIM. Sliding the scroll bars causes the contents of the window to scroll, with RefreshPart called automatically to redraw newly-exposed parts of the window.

RefreshPart routines for practical programs are longer, but not complex. Fig. 5 shows the RefreshPart routine for our DTM placement and routing program.

Most applications also have an "InputEvent" function which interprets interactive input
fn DTMrefresh =
(var p: GMDSptr;
 GMDSbase:= SelWin.data: GMDSptr;
 if show routes then PRdraw trouble;
 if show rats then SZdraw rats nest;
 SZdraw grid;
 if show PEC > 0 then PRdraw PEC;
 if show overlaps then SZdraw overlaps;
 if show occ then SZdraw occ;
 -- Draw blocks --
p:= GMDSbase; ForeColor:= 1;
 repeat
  if p.opcode = CALLX then
    (var (z1,z2,zc): Pos;
    z1:= [p.Cpos.x * scale, p.Cpos.y * scale];
    z2:= z1 + [Word (p.Csize.x << 2) * scale,
               Word (p.Csize.y << 2) * scale];
    draw rect from z1 to z2);
  p:= next p
until p.opcode = RETURN$;
if show routes then (PRdraw routes; PRdraw route endpoints);
if show ECH then SZdraw ECH;
if show force then SZdraw force;
ForeColor:= 1 -- Default color = Black --
)

Figure 5: RefreshPart routine for DTM placement and routing program.
events such as mouse clicks and key presses. G-SWIM calls this function when user input occurs. The application then takes appropriate action, often modifying application data structures and displaying the results in the window. A complete description of G-SWIM and larger examples can be found in [2].

The Digital Transform Machine

The MITRE Digital Transform Machine [5] is a reconfigurable logic array intended for gate-level emulation of high-performance signal and image processing hardware. DTM consists of variable-size programmable logic blocks with PLA-like form, interconnected by a hierarchy of 2-D meshes which connect cells that are $2^p$ cells apart. Fig. 6 shows part of a DTM with ten blocks of various sizes. The numbers indicate how cells can be interconnected: a cell labelled $k$ can connect to its four nearest neighbors also labelled $k$.

The DTM placement and routing problem consists of (1) placing non-uniform $M \times N$ blocks of cells within the array, avoiding overlaps, and (2) connecting the blocks subject to the constraints of the connection meshes. As with most integrated circuits, it is fruitful to partition the problem into separate placement and routing phases. The placement phase is designed to place blocks to ensure high-quality routing. The routing phase assumes the blocks are immobile.

Both placement and routing were prototyped using Galaxy. The next sections describe
Figure 7: Placement visualization: rat's nest connections and edge-crossing histograms.

how visualization was used to observe and improve algorithm performance.

DTM Placement

Our DTM placement algorithm uses a new variant of Force-Directed placement [10] which considers both attraction and repulsion. Force-directed placement models the connections between blocks as spring forces — connected blocks are attracted to each other. Fixed points at the perimeter of the array — i.e., external pins — prevent the blocks from coalescing at a single point. The classical algorithm allows arbitrary overlaps in the resulting placement. These must be corrected in a subsequent assignment phase.

A number of variations were tested and incorporated in DTM placement (DTM-P). These were all implemented as forces. In addition to classical spring forces, DTM-P considers a diffusion force which pushes blocks away from congested areas of the chip. Second, DTM-P incorporates an anti-overlap force to separate overlapping blocks in place of a separate assignment pass. The anti-overlaps force is very effective for variable size blocks. Third, a random force is added to help prevent local minima. This produces some of the benefits of Simulated Annealing [7] without its high computational cost.

To simplify computations, the $x$ and $y$ components of each force are calculated separately. The components are then scaled by weights for each type of force and added together. After the vector force for each block has been computed — preferably, in parallel — then
all blocks are moved simultaneously in proportion to the magnitude and direction of their forces. The blocks are modelled as if in a viscous medium, so they are limited by a "terminal velocity" proportional to their forces.

The relative weights of the four forces and the magnitude of the velocity have a large impact on the quality of the final placement. If the spring force is too large, the blocks will cluster together. If too small, connected blocks do not cluster well. If the velocity is too small, the algorithm will run very slowly. If the velocity is too large, blocks will leap over each other, and even oscillate.

A visual display was implemented to explore the effects of the forces on blocks. A window displays the current placement of blocks in the array along with calculated data which determines the next placement, shown graphically. These include the $x$ and $y$ diffusion profiles for calculating diffusion force, the $x$ and $y$ edge-crossing histograms [7] for calculating spring forces,² the "rat’s nest" of connections between blocks, block overlaps, and forces on components. Each kind of data is shown in a different color, and can be displayed or left hidden as desired. For example, Fig. 7 shows blocks, "rat’s nest" connections, and edge-crossing histograms.

The individual and total forces on blocks can also be shown, in the form of vectors. Fig. 8 shows forces due to overlapping blocks. The size of the square in each cell indicates the degree of overlap. Blocks are actually considered to overlap themselves, but this does

²Edge-crossing histograms count the total number of connections crossing array rows or columns.
Figure 9: DTM-P control panel: buttons and sliders control placement parameters.

not affect net force since the overlap forces on each side of the block balance. True block overlaps result in large, unbalanced overlaps which produce the force vectors originating from the center of each block.

The user can change the relative weights of the forces using sliders in a “control panel” (Fig. 9) and see immediately how this affects the magnitude and direction of the total forces. Forces can be disabled completely by setting their weights to zero. The magnitude of velocity can also be set.

The visual feedback from these displays quickly refined DTM-P. We found that the following algorithm worked quite well:

1. Diffusion: turn on the diffusion force and turn off the anti-overlap force, and set the velocity high — allow blocks to move several cells each iteration. This obtains a good distribution of overlapping cells.

2. Anti-overlap: turn off the diffusion force and turn on the anti-overlap force, with the velocity low. This causes overlapping cells to move apart.

We also found that detecting actual block overlap was not sufficient: we needed to create a lower-weight guard ring around each block to prevent overlaps. The need for this became quite clear by observing visual displays of cell overlap like Fig. 8.

Random forces proved to be very important for avoiding local minima. Without them, small blocks would not move into available holes if the path to them was obstructed. Random forces also cause overlapping components to separate more quickly.

Another important addition to placement was to allow blocks to be misaligned with respect to the cell grid during most of the placement. This allowed small movements of a block to accumulate over several passes. With coarse alignment, the forces would never be strong enough to move the block by a whole grid point.
DTM Routing

Once the placement is complete, routing can begin. The DTM uses an interconnection network called Packed Exponential Connections (PEC) [6]. A cell can be connected to cells that are a multiple of $2^p$ away in any of the four directions, where $p$ is distributed as shown in Fig. 6. Half the cells have distance $2^1$ connections, 1/4 have distance $2^2$ connections, 1/8 have distance $2^3$, etc. The exponent $p$ of a cell is called its PEC number.

Connections are made to block rows and columns. The cells in a row or column are electrically connected so external connections can be made to any cell of a row or column. In addition, the rows and columns of a block can be permuted without changing logic function. It does not matter which row or column is connected as long as each connection is to a distinct row or column.

The DTM-R routing algorithm uses Dijkstra's shortest path algorithm [4] to route point-to-point connections. To connect a source block $S$ to a target block $T$, we find the lowest cost path from any of the source's cells to any of the target's cells. This is accomplished by a best-first search through a graph model of PEC. Dijkstra's algorithm guarantees an optimal solution for a single route. Routes must always have L, Z, or U shape, which limits the search space. The resulting route is through cells with the same PEC number, though this constraint can be relaxed.

It is highly unlikely that DTM-R could route all wires in a single pass, so it uses Penalty-Driven Iterative Improvement (PDII) [8, 9]. PDII makes multiple passes through the net.
list, routing each wire using the optimal point-to-point algorithm. On the first pass, the first nets routed find excellent paths since they are unblocked. Later nets often require illegal overlaps. PDII penalizes rather than forbids illegal routes, so that routing can complete.

On each subsequent pass, every wire is ripped up and rerouted. When the first wires are rerouted, they are subject to the blockages caused by later wires. They therefore find alternative routes that avoid blockages at the expense of increased length. This opens up paths for the later nets. After several passes, the paths are distributed over the array and illegalities become rare or non-existent.

Visual display of routes was very important for debugging and optimizing the routing algorithm. PEC allows very dense routing: the maximum local density of a row or column is the maximum PEC number of the array (6 for a $64 \times 64$ array). To display routes effectively, we draw paths in parallel vertical and horizontal tracks, with each PEC number assigned to its own track, as shown in Fig. 10. This ensures that routes do not overlap except for illegal routings.

In addition, the routes for different PEC numbers are drawn using different colors on a color monitor. This makes it much easier to follow routes from block to block, which can be compared to the rat’s nest. Colors also show at a glance how well each PEC number is utilized. In the current version of the router, PEC(2) and PEC(3) are the most utilized. PEC(1) routes require too many hops, and PEC(4) and higher require certain block alignments. Utilization is a function of the average size of blocks: larger blocks will be able to use larger PEC numbers.
Users can also see the PEC numbers, displayed on top of the placement and routing in the style of Fig. 6. The PEC numbers are shown in the same colors as the routes, making it easy to see their relationships. As an option, users can look at just the PEC numbers associated with actual routes, as shown in Fig. 11. This shows very clearly how many hops are required for each segment of a path.

Illegal routes are shown visually by filling bad cells with yellow before drawing the rest of the data. These are common after the first PDII pass (Fig. 10), but most or all disappear in subsequent passes (Fig. 11). The yellow cells clearly show trouble spots and helped the developer determine how to fix them.

Summary

Rapid prototyping of visualization tools was essential for developing DTM placement and routing. It was not at all obvious at the beginning what the best algorithms would be. The visualization tools clearly showed the effect of the different forces on the placement process and the effectiveness of the routing algorithm.

Visualization tools such as this can only be effective if the necessary GUI can be rapidly prototyped. The Galaxy Programming Environment proved to be highly productive for constructing DTM-P, DTM-R, and their visualization tools. The entire prototyping process took the equivalent of three weeks of full-time effort.

Acknowledgements

This work was performed by the author while he was with the University of Wisconsin–Madison. He would like to thank Worth Kirkman of The MITRE Corporation for introducing him to DTM and providing technical details necessary for designing the algorithms.

This material is based upon work supported by the National Science Foundation under Award No. MIP-9102382. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

References


3Duplicated in black on Figs. 10 and 11.


