平成 7 年度

筑波大学第三学群情報学類

卒業研究論文

題目：ビジュアルプログラミングシステムのための undo 機能の研究

主専攻	情報科学
著者名	馬場 昭宏
指導教員	電子・情報工学系 田中 二郎

要 旨

人間が何かを行なう時、間違いが発生することがある。これらの間違いを修正するのに最も簡単な方法はその行なった動作を取り消すこと、すなわち undo である。

ビットマップディスプレイはウインドウシステムを始めとするユーザインタフェースを発達させた。このため、専門家以外の多くの人がコンピュータを使うようになった。

一方、ビットマップディスプレイは機能の複雑化を促したため、 undo 機能は更に重要になった。しかし、従来の undo 機能には無駄が多かった。

これらの事情により、初心者にも使いやすく、かつ、使い慣れた人にとってはより短い時間で undo できるようなシステムが望まれている。

本論文では、木構造を用いた undo モデルと、ミニチュアを用いたインタフェースを提案し、それらを実際に用いてビジュアルプログラミングシステム PP のために作成した undo システムについて述べる。

目 次

第 1 章

はじめに

1.1 エラーに対応するための方法

コンピュータは道具の中でもっとも複雑なものの一つである。このことはそのヒューマンインタフェースに関する部分についても言える。

はさみや鉛筆などの通常の道具では、その外観などからそれをどのように使えばよいのかがわかり、間違った用い方をされることは少ない。

ところがそれら通常の道具から電気製品、コンピュータと複雑化していくにつれ、それが何をするためのものか、どのように用いればよいのか、ということがわかりにくくなり、エラーを犯しやすくなってしまった。これらのエラーは防止されなければならない。

エラーを犯しにくくするためのよいデザインの原則として Norman は次のようなものをあげている [1]。

可視性 目で見ることによって、ユーザは装置の状態とそこでどんな行為をとりうるかを知ることができる。

よい概念モデル デザイナーは、ユーザにとってのよい概念モデルを提供すること。そのモデルは操作とその結果の表現に整合性があり、一貫的かつ整合的なシステムイメージを生むものでなくてはならない。

よい対応づけ 行為と結果、操作とその効果、システムの状態と目に見えるものの間の対応関係を確定することができること。

フィードバック ユーザは、行為の結果に関する完全なフィードバックを常に受け取ることができる。

これらはデザインする上での原則であるが、システムメッセージやオンラインヘルプを充実させることもエラーの防止に役立つ。ユーザの不確実な知識を確実なものにしてくれるからである [2]。

しかしどんなによいインタフェースを用いても必ずエラーは起きる。したがってエラーを修正するための方法を考えなければならない。

エラーを修正するための方法として逆操作がある [3]。１次元の文字列を扱う line editor においては、逆操作は単に文字を消去することだけであった。だが時代が進み screen editor になると全体を見渡せることを活用したさまざまな機能が付加され、それにともなってさまざまな逆操作が必要となったため、ユーザはどのような逆操作をすれば元の状態に戻すことができるのかを考えなければならなくなった。

元に戻す方法がわからない初心者のユーザにとっては、このことは重要な問題となる。なぜならユーザがアプリケーション (ここでは editor) を使う方法を覚えるための近道は試行錯誤することだからである。いざというときに元に戻せないとユーザは失敗を恐れていろいろ試すことができなくなってしまう。

undo(取消) 機能が提供されていれば、ユーザは元の状態に戻す手順を考える必要がなくなるためさまざまな機能を試してみることができる。

Emacs[4] では任意の回数だけ元の状態に戻すことのできる undo 機能を提供している。通常、１つのコマンドが使われると、変更取消し記録とよばれる記憶域に項目が１つ作られる。ただし、コマンドによっては１つのコマンドで複数の項目が作られたり (query-replace など)、複数のコマンドで１つの項目が作られる (自己文字挿入など)。 undo コマンドを１回適用すると、１つ前の状態に戻る。これを全ての変更取り消し記録が取り消されるまで続けることができる。

ほかのコマンドを使用すると、 undo コマンドの連続を中断する。一度中断するとこれまでの undo コマンド自身が取り消し可能な普通のコマンドと見なされる。 Emacs ではこのことを利用して undo の undo を行なう。

変更取消記憶は各バッファごとに作られる。従って複数のファイルを同時に編集していても undo は現在編集中のバッファに対してだけ行なわれる。

実際に undo していく様子を図 1.1に示す。

ビットマップディスプレイを用いた近年の環境においては元の状態に戻すための手順はより複雑になるため、 undo 機能はこれまでよりもさらに重要なものとなっている。

Macintosh[5] では、ほとんどの機能をメニューから選択する形式で操作できる。 undo 機能も同様である。処理の進行状況によりメニューアイテムの undo の表示が変化し、その時点で undo 可能な内容がわかるようになっている。例えば編集メニュー (Edit menu) での” 取り消し” (Undo) の表示は、” 入力取り消し” (Undo Typing)、” カット取り消し” (Undo Cut) などである。最後に行われた操作が” 取り消し” (Undo) であった場合、メニューアイテムの表示は” 取り消し” から” 〜を元に戻す” に変わる。” 〜” の部分には取り消された操作内容が表示される。ここで” 〜を元に戻す” を選ぶと、” 取り消し” の操作自体が取り消される。

入力	番号	状態	画面	ヒストリ
	0	0	`abcd`	0
delete	1	1	`abc`	01
delete	2	2	`ab`	012
delete	3	3	`a`	0123
undo	4	2	`ab`	01232
undo	5	1	`abc`	012321
`e`	6	4	`abce`	0123214
undo	7	1	`abc`	01232141
undo	8	2	`ab`	012321412
undo	9	3	`a`	0123214123
undo	10	2	`ab`	01232141232
undo	11	1	`abc`	012321412321
undo	12	0	`abcd`	0123214123210

図 1.1: Emacs での undo の様子

第 2 章

木構造を用いた undo

2.1 ヒストリを単に戻るだけの undo の煩わしさ

エラーには大きく分けてスリップ (slip) とミステイク (mistake) の 2 種類がある [1]。スリップは何かを実行する時に起こるエラーであり、この場合大抵 2、 3 ステップ程度 undo すればよい。一方、ミステイクは実行するための目標を立てる時点でのエラーである。目標が間違っているとそれまで実行してきたものは実際にしたいこととは根本的に異なるため、かなりのステップ数 undo しなければならない。

図 1.1の例では、状態 4 から状態 0 にするまでに 6 ステップかかっている。これは undo 操作自身がヒストリに残っていることが原因である。実際に使用している時には状態の数はこの例よりもはるかに多くなるり、 undo もより多く使われるため、目的の状態に戻るまでにはより多くのステップ数を必要とする。これでは同じ状態を行ったり来たりしているようでフラストレーションを感じる。

Tgif[6] ではこの問題を解決するために undo と redo によりヒストリ中を移動するという方式を採用している。 Tgif で undo をして行く様子を図 2.1に示す。状態 0、 1、 2、 3、 4 は図 1.1の各状態に対応している。

番号 0 から番号 2 の間はヒストリには次々に新しい状態が付け加えられていく。番号 3 から番号 5 まではヒストリ自身には変化はない。変化するのは、その中での現在の状態へのポインタである。図 2.1では太字で表したものが現在のポインタの位置である。 undo をすることでポインタが左 (古い方) に、 redo をすることでポインタが右 (新しい方) に動く。番号 6 では状態 4 が作られ、その時ヒストリからはポインタより後の状態 2 と状態 3 は削除され、状態 4 が加えられる。

番号	状態	ヒストリ
0	0	**0**
1	1	0**1**
2	2	01**2**
3	3	012**3**
4	2	01**2**3
5	1	0**1**23
6	4	01**4**
7	1	0**1**4
8	0	**0**14

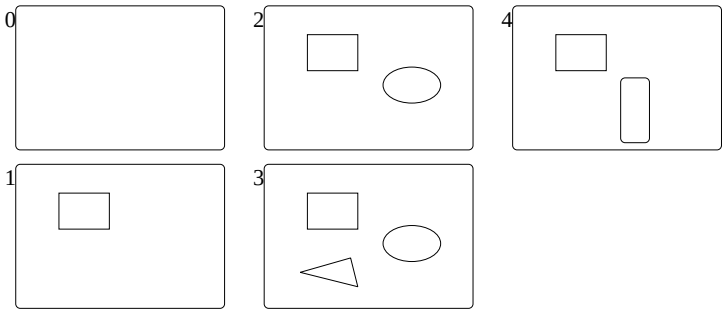


図 2.1: tgif での undo の様子

2.1.1 状態の木

Tgif の undo 機能は Emacs と比較してより少ないステップ数で前の状態に戻ることを可能としている。しかし undo してから新たな状態を作ると、図 2.1に示したように、それまでに作った状態は消されてしまう (図 2.1)。これではユーザは元の状態に戻れるという前提のもとに試行錯誤を重ねることができなくなってしまう。そこでこれらの状態を削除せずに残しておくことを考えると、状態の集合は 1 つの木構造を形成する (図 2.2)。このようにしてできた木構造を状態の木と呼ぶことにする。

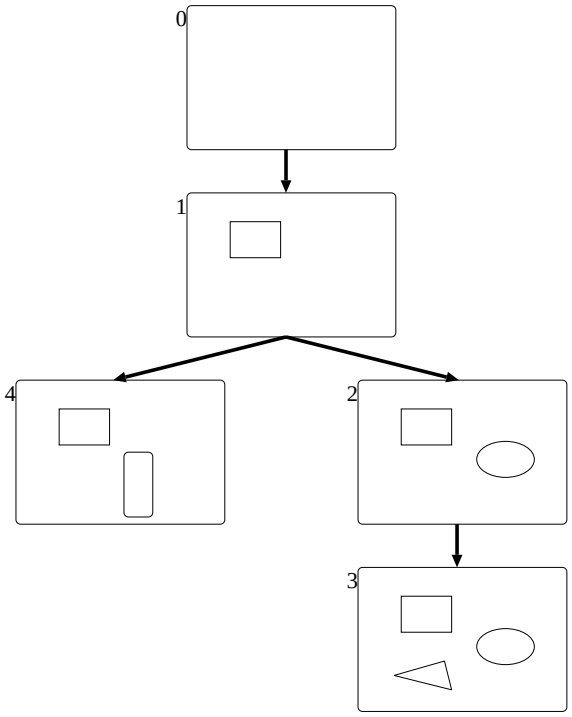


図 2.2: 状態の木

第 3 章

PP の undo 機能

本章では今回作成した undo システムの機能について述べる。

本システムはビジュアルプログラミングシステム PP の定義節エディタ [7][8](図 3.1) の一部として作成した。

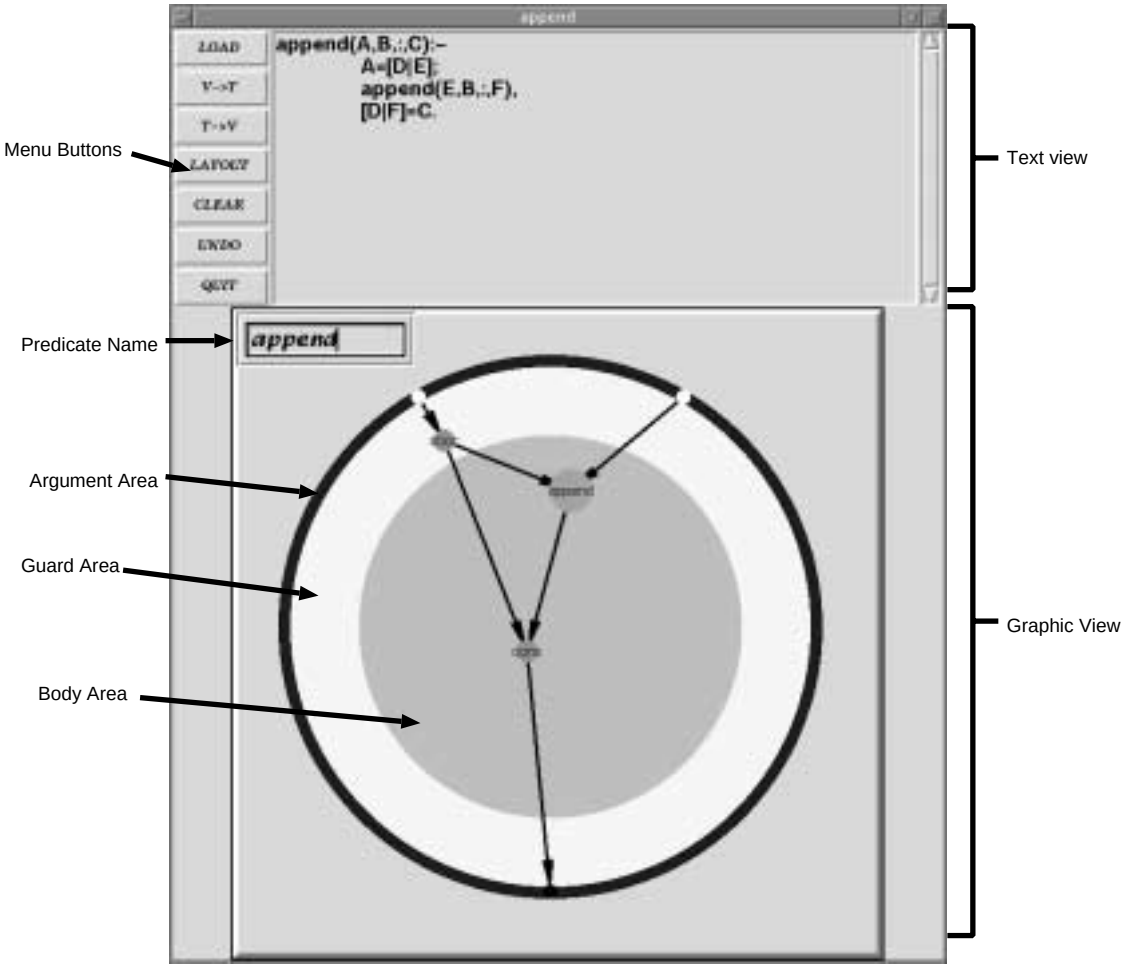


図 3.1: PP

3.1 状態

PP の中でなにか入力があると新しい状態が作られ、保存される。保存される状態の数の上限はユーザが決めることができる。

状態の数が上限に達した時は削除される。削除の対象となるのは状態の木において現在の状態から最も遠いもので、候補がいくつかある場合はそれらのうちで最も古いものである。

3.2 モデル

本システムで用いられている undo のモデルは 2 つある。 1 つはヒストリを用いたもので、もう 1 つは木構造を用いたものである。

3.2.1 ヒストリ

ヒストリを用いた undo は Emacs のものと似ている。新しい状態が作られた時、または undo をしたときのいずれかの場合にヒストリには現在の状態が付け加えられる。 Emacs のものと異なるのは、ヒストリを戻るだけでなく、進むことができるという点である。これにより、 Emacs のように他のコマンドを入れて undo を一旦中断することで undo しすぎた時の取消をしなくてもよくなった。

ヒストリ中を新しい方へ進んだ時にもヒストリに現在の状態が加えられると、ヒストリに無限列が生じてしまう可能性がある (図 3.2)。図 3.2で太字で表されているのは現在の状態である。無限列が生じてしまうと、ユーザは最新の状態がどれなのかわからなくなってしまうおそれがある。本システムではこの問題を回避するために、一種のストッパーを用いている。図 3.3で大きめの字で表されているのがストッパーの位置である。ヒストリ中の移動はストッパー以前のものに制限される。新しい状態が作られると、ストッパーはヒストリ中のその状態の位置に移動する。

	012**3**
戻る	01**2**32
進む	012**3**23
進む	0123**2**32
進む	01232**3**23
⋮	⋮

図 3.2: 無限列の例

012**3**
戻る　01**2**32
進む　012**3**23
これ以上進めない

図 3.3: ストッパーを用いた場合

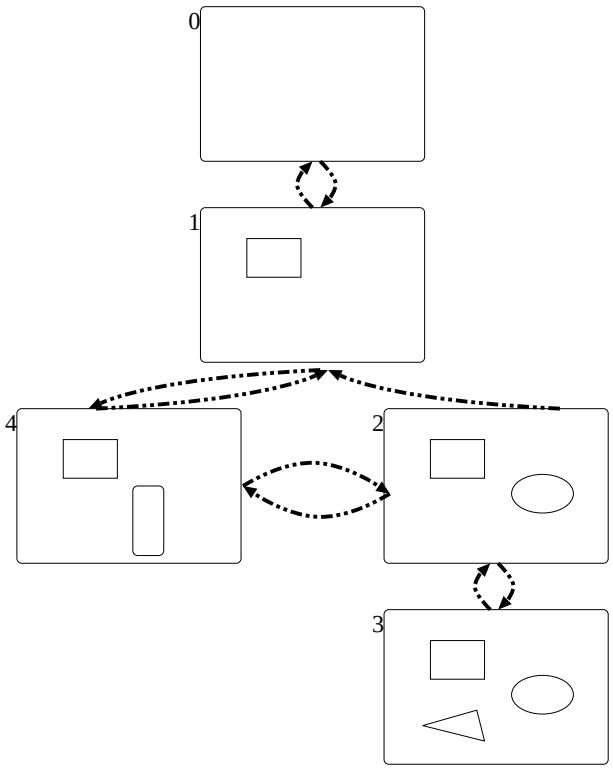


図 3.4: 実際の構造

3.2.2　木構造

本システムでは状態の木の中を移動するために 4 つの方向を用いている。親、最も若い子、すぐ隣の兄、すぐ隣の弟である。具体的にどのような構造になるかを図 2.2の木構造を例に用いて示す (図 3.4)。図 3.4中で矢印は、その始点のノードから、終点のノードへ移動できるということを意味している。例えば状態 1 から移動できるのは親 (0) と、最も若い子 (4) であり、状態 2 から移動できるのは親 (1)、すぐ隣の弟 (4) および最も若い子 (3) である。

子どものうちで最も若いものが選ばれる理由は、それがユーザが現在意図している状態に近いものであると考えたからである。

親と、最も若い子の間でのみ移動可能であるように限る、つまり兄弟間の移動をなくすと、 Tgif の undo と同じになる。

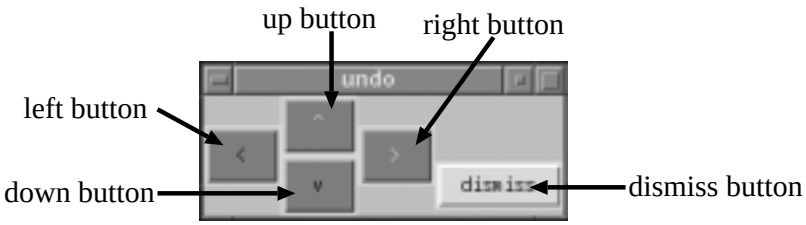


図 3.5: cross

3.3 インタフェース

本システムでは cross、 list、 treeGraph、 linear、 linearGraph、 allGraph、 allGScrl、 selectGraph の 8 種類のインタフェースを用意している。これらは主に 3.2節で示した 2 つのモデルを用いている。

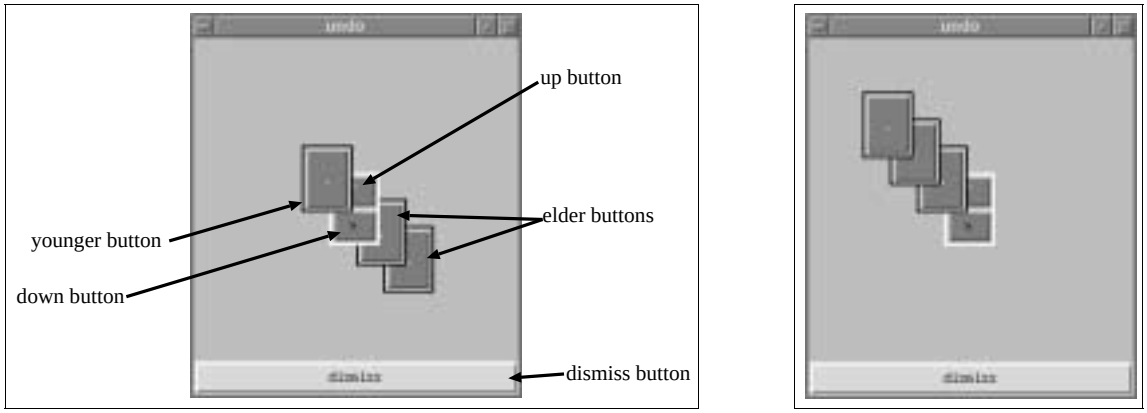
これらのインタフェースは 1 枚のウインドウとして表示される。今後、これらのウインドウのことを undo window と呼ぶことにする。 undo window は PP のメニューボタンの中で UNDO を選ぶことによって 1 枚だけ開かれ、 undo window の中の dismiss ボタンを押すことによって閉じられる。

3.3.1 cross

cross は木構造のモデルを直接用いたインタフェースである (図 3.5)。上、下、左、右のボタンにより、それぞれ、親、もっとも若い子、すぐ隣の弟、すぐ隣の兄の状態に移動する。それらの方向に移動できない時はボタンの矢印は黒で、移動できる時は黄色で表示される。

3.3.2 list

list も木構造のモデルを用いているが、現在の状態の兄弟すべてに直接移動できるようになっている。図 3.6(a) の中央付近の白くふちどりされた 2 つのボタンの上ボタンで親に、下ボタンで最も若い子に移動する。その 2 つのボタンより右下にならんでいるボタンが兄、左上にならんでいるボタンが弟への移動に使われる。これらのボタンは現在の状態を常に反映している。例えば図 3.6(a) で最も右下のボタンを押すと 2 つ上の兄の状態に移動し、 undo window は図 3.6(b) のように変わる。



(a) (b)

図 3.6: list

3.3.3 treeGraph

treeGraph も cross と同様、木構造のモデルを直接用いている。中央の current と書かれたものの上、下、左、右に表示されている PP のグラフィック・ビューのミニチュアがそれぞれ、親、最も若い子、すぐ隣の弟、すぐ隣の兄の状態である (図 3.7)。親、子、兄、弟がない場合にはそれらは表示されない。ミニチュアの中にマウスカーソルが入ると、ミニチュアの枠がハイライト表示され、そこでマウスの左ボタンをクリックするとミニチュアによって表される状態に移動する。このことはミニチュアを用いたインタフェースすべてにおいて共通である。

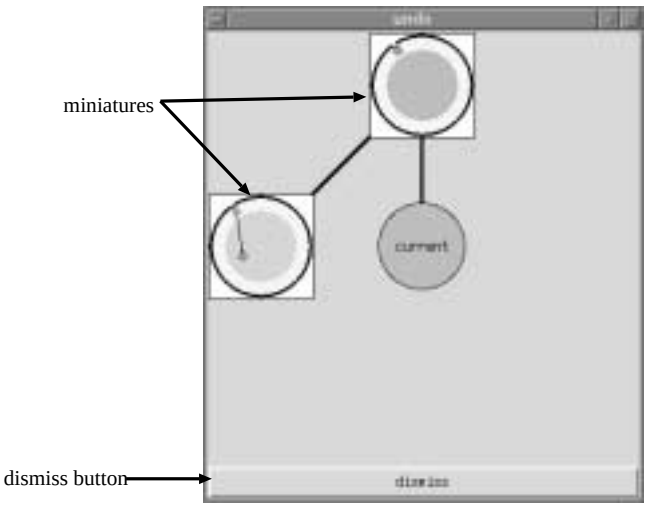


図 3.7: treeGraph

3.3.4 linear

linear はヒストリモデルを直接用いたインタフェースである (図 3.8)。 prev ボタンでヒストリを一つ古い方へ進み、 next ボタンで新しい方へ進む。それらの方向に移動できない時はボタンの文字は黒で、移動できる時は黄色で表示される。

図 3.8: linear

3.3.5 linearGraph

linearGraph も linear 同様、ヒストリのモデルを用いたインタフェースである。中央の円の中に current と書かれたものの左にヒストリ中の一つ前のミニチュアが、右に一つ後のミニチュアが表示される (図 3.9)。

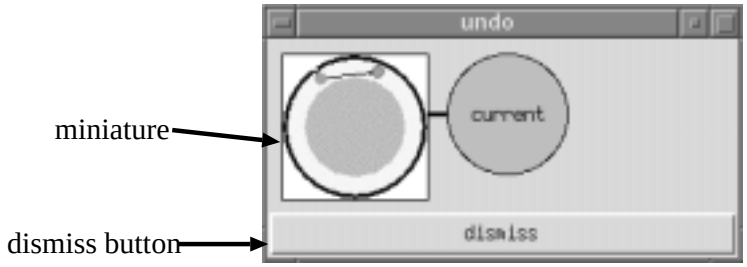
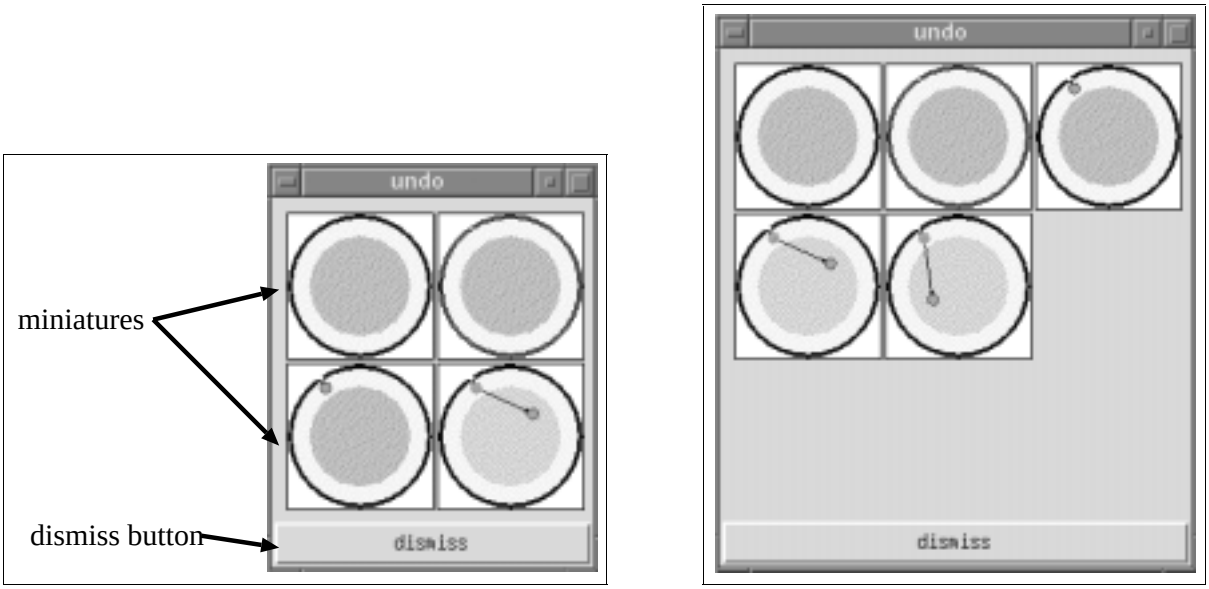


図 3.9: linearGraph

3.3.6 allGraph

allGraph はとくにモデルを用いていないインタフェースで、現在あるすべての状態のミニチュアが順番に表示される (図 3.10)。 undo window の大きさはすべての状態を表示し得る最小の正方形領域が確保され、状態の数が変化するのにしたがって動的に再配置される。図 3.10(a) では状態の数は 4 つだが図 3.10(b) では状態の数が 5 つになり、表示しきれなくなったのでより大きな領域が確保され、配置も変化している。



(a) (b)

図 3.10: allGraph

3.3.7 allGScrl(allGraph with Scrollbar)

allGraph では状態の数を保存しておける数の上限があまり大きいと undo window が画面に収まりきれなくなってしまう可能性がある。そこで、スクロールバーを設けることでこの問題を解決したのが allGScrl である (図 3.11)。

allGScrl と 3.3.8節で示す selectGraph では undo window の大きさをあらかじめ設定しておく。状態の数が増えてその大きさを越えた場合は allGScrl ではスクロールバーを用いて目的の状態を探すことができる。

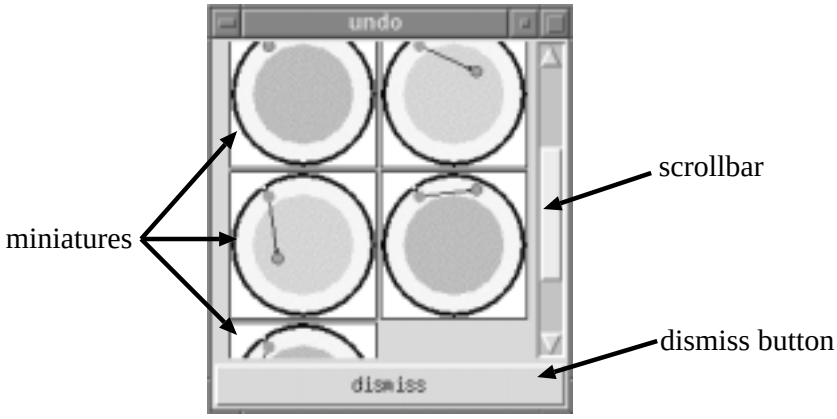


図 3.11: allGScrl

3.3.8 selectGraph

selectGraph(図 3.12) では状態を飛び飛びに表示させることでスケーラビリティの問題を解決している。ただし、すべての状態が表示されるわけではないため、他の手段を併用する必要がある。

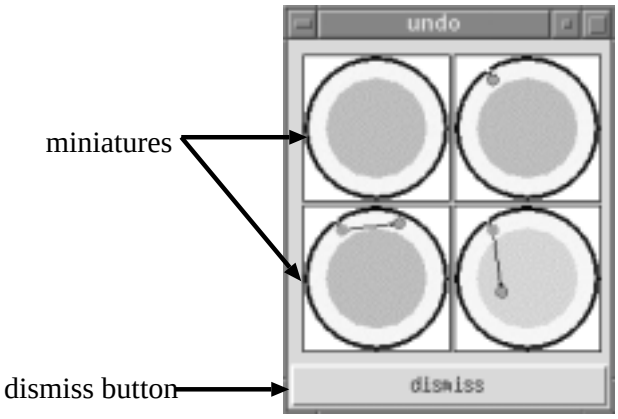


図 3.12: selectGraph

3.4 カスタマイズ

ユーザの要望はさまざまであるため、本システムではいくつかのことがらをカスタマイズできるようになっている。

ユーザは次のようなことを環境変数、または設定ファイルを用いて変更することができる。同じことがらについて環境変数と設定ファイルの両方で設定した場合は環境変数の方が優先的に用いられる。

- undo の種類 (3.3節で述べた 8 種類)(`undo_type`)
- 保存しておける状態の数 (`maxStateNum`)
- allGScrl、 selectGraph などで表示できる状態の最大数 (`wid`)
- ミニチュアを作成するかどうか (1: 作る それ以外: 作らない)(`miniatures`)

例えば環境変数を用いて undo の種類を selectGraph に設定する場合、シェルのコマンドラインで次のように入力する。

```
% setenv undo_type selectGraph
```

設定ファイルを用いる場合、設定ファイル `.pp` 中で設定されている各変数の値を書き換える。例えば、保存しておける状態の数を 200 に変更する場合、`.pp` の次の行

```
set maxStateNum 100 ;# Saves states until this number
```

を

```
set maxStateNum 200 ;# Saves states until this number
```

と書き換えればよい。

環境変数や設定ファイルでユーザが使用不可能な値を設定した場合、デフォルトの値が用いられる。

第 4 章

実装

以上のような機能を、 Tcl/Tk[9] を用いて実装した。

4.1 状態の内部表現

n1 つの状態は次のような Tcl の連想配列`state` である。

`state(up,`*num*`)`	状態 *num* の親の状態の番号
`state(down,`*num*`)`	状態 *num* の最も若い子の状態の番号
`state(left,`*num*`)`	状態 *num* のすぐ隣の弟の状態の番号
`state(right,`*num*`)`	状態 *num* のすぐ隣の兄の状態の番号
`state(str,`*num*`)`	状態 *num* の内部コード (4.3節参照)

例えば図 2.2のような状態の木ができている時には表 4.1のようになっている。

num	up	down	left	right
0	-1	1	-1	-1
1	0	4	-1	-1
2	1	3	4	-1
3	2	-1	-1	-1
4	1	-1	-2	2

表 4.1: 状態の木の内部表現

4.2 ヒストリの内部表現

ヒストリは状態の番号を要素として持つ Tcl のリスト `history` である。現在ヒストリのどこにいるのかを指している変数 `hisp` と、ストッパーの役割をする変数 `redop` を用いている。

4.3 PP の内部コード

PP での内部コードは次のようになっていた。

{定義節のゴールのコード {ガード部のゴールのコードのリスト} {ボディ部のゴールのコードのリスト}}

ゴールのコード

{ゴールの名前 / 種類 / 番号 {入力引数のコードのリスト} {出力引数のコードのリスト}}

引数のコード

引数の名前 / 種類 / 番号

しかし、この方式だとグラフィック・ビューでの各ゴール、引数の位置を表す情報を記録することができないため、今回これに位置を追加し、次のようにした。

ビジュアルから作られた (v) かテキストから作られた (t) か {定義節のゴールのコード {ガード部のゴールのコードのリスト} {ボディ部のゴールのコードのリスト}}

テキスト表現から作られた場合ゴールのコードと引数のコードは以前と同じだが、ビジュアル表現から作られた場合は次のようになる。

ゴールのコード

{ゴールの名前 / 種類 / 番号 /x 座標 /y 座標 {入力引数のコードのリスト} {出力引数のコードのリスト}}

引数のコード

引数の名前 / 種類 / 番号 /x 座標 /y 座標

4.4 状態の削除

状態は 4.1節で述べたように配列 state で表されているが、ミニチュアが作成されている場合にはこの他に、ミニチュア (4.5節) の削除も行なう必要がある。

新しい状態を作る時には、現在の状態の数と、保存可能な状態数の最大値 (maxStateNum) とを比較し、前者が後者を越えた場合には 3.1節で述べた基準にしたがって削除される。現時点ではこの基準で自動的に削除されているが、葉ノード以外のノードを選んでその子孫をすべて削除するように実装されているので、ユーザインタフェースさえ備えればユーザが任意のノード（およびその子孫）を削除できる。

4.5 ミニチュア

Tk4.0 からは canvas ウィジェットのなかに image と呼ばれるものを表示できる。 image は `image` コマンドによって作成され、 GIF、 PPM/PGM 形式が使える。

また、 canvas ウィジェットは現在の状態を PostScript 形式で出力する機能がある。

`gs` は Ghostscript のインタプリタ／プレビュアであるが、 PostScript 形式のデータを PBM、 PPM などの形式に変換するためのフィルタとしての機能も持っている。

本システムではこれらを利用してミニチュアを作っている。まず、 PP のグラフィック・ビューから PostScript 形式で出力する。 sub process として `gs` を起動し、 `gs` 中で PPM 形式に変換、更に縮小する。これを `image` コマンドで取り込む。

4.6 新しいインタフェースを作るための機能

本システムでは 3.3節で述べたように 8 種類のインタフェースを備えている。しかしこれらのインタフェースをユーザが気に入らない場合、新しいインタフェースをユーザ自身が Tcl/Tk を用いて作りやすいよう、ミニチュアの表示、ヒストリや状態の木の中の移動などのいくつかの手続きを用意している。

各インタフェースは基本的には次のようになっている。

① undo window の表示など、初期化

② ボタンやミニチュアなどの書き換え。これは視覚的なものだけでなく、ボタンやミニチュアをクリックした時に呼び出されるコマンドの書き換えも含む。それらのコマンドのほとんどは移動コマンドである。

③ ボタン、ミニチュアがクリックされると bind されていた移動コマンドが呼び出される。移動コマンドは移動の他に、移動した後に②を行なう。

実際に提供しているコマンド群は次のものである。

表示関係

`printstate` *num can x y tag move_command*

機能： ミニチュアの表示、移動コマンドの bind をする。

引数：

num	表示する状態の番号
can	表示したい undo window の名前
x	ミニチュアの左上の点の x 座標
y	ミニチュアの左上の点の y 座標
tag	このミニチュアを識別するための tag 名
move_command	このミニチュアに bind する移動コマンド

戻り値： なし

`clean_up_undo_window` *can*

機能： undo window 中のミニチュアをすべて消す。

引数： *can* この名前の undo window からミニチュアが消される。

戻り値： なし

移動関係

`move` *dir change_button_command*

機能： 木構造モデルを用いて移動する。

引数： *dir* `up`、`down`、`left`、`right` で状態の木中の親、最も若い子、すぐ隣の弟、すぐ隣の兄に移動する。

change_button_command 移動後に undo window を書き換えるためのコマンド

戻り値： なし

`next` *change_button_command*

機能： ヒストリモデルを用いて新しい方へ移動する。

引数： *change_button_command* 移動後に undo window を書き換えるためのコマンド

戻り値： なし

`prev` *change_button_command*

機能： ヒストリモデルを用いて古い方へ移動する。

引数： *change_button_command* 移動後に undo window を書き換えるためのコマンド

戻り値： なし

`move_by_num` *num change_button_command*

機能： 番号を指定してその状態に移動する。

引数： *num* 移動先の番号

change_button_command 移動後に undo window を書き換えるためのコマンド

戻り値： なし

undo 関係で用いられる主な global 変数には次のようなものがある。

`curp`	: 現在の状態の番号を保持
`stateNum`	: 現在ある状態の総数
`maxStateNum`	:3.4節参照
`wid`	:3.4節参照
`state`	:4.1節参照
`history`,`hisp`,`redop`	:4.2節参照

これらを用いてファイル `undo.tcl` 中に次の手順で新しいインタフェースを追加する。

1. ファイル `main.tcl` 中の変数`_all_of_undo_types` に新しく作るインタフェースの名前を加える。そのインタフェースがミニチュアを使う場合は`_undo_types_using_Graph` にも加える。

2. ①を行なう手続き `undo`*num* を作る。 *num* は`_all_of_undo_types` の中での新しく付け加えたインタフェースの名前の順番である。 `undo`*num* のひながたは `undo.tcl` に入っているので、それをコピーして使う。

3. ②を行なう手続き`change_button`*num* を作る。 *num* は 2 と同じものである。

第 5 章

評価

5.1 効率性に関する定量的評価

ヒストリモデルと木構造モデルに関して、実際にどの程度木構造モデルが効率的であるかの評価を行なった。

5.1.1 測定の対象

状態数 n の場合の状態の木は $(n-1)!$ 通り存在し得る。

例えば、状態数が 4 の場合、図 5.1の枠で囲った 6 種類の状態の木 A ～ F とそれに付随するヒストリがある。

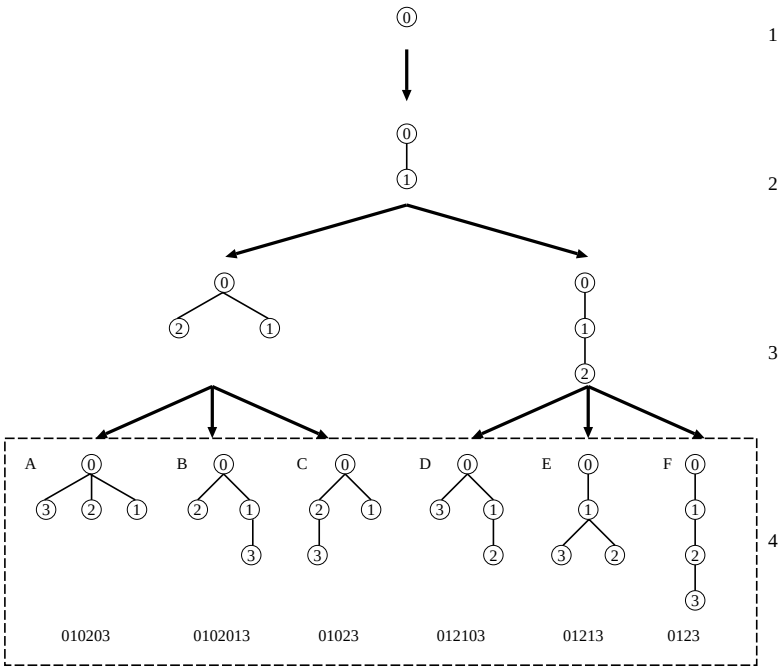


図 5.1: 状態数 4 の場合の状態の木

木構造

	0	1	2	3
0	0	2	3	1
1	1	0	1	1
2	2	1	0	2
3	1	1	2	0

ヒストリ最小

	0	1	2	3
0	0	1	2	1
1	1	0	1	2
2	2	1	0	3
3	1	2	3	0

ヒストリ平均

	0	1	2	3
0	0	1	2	3
1	1	0	1	3
2	2	1	0	3
3	1	2	3	0

ヒストリ最大

	0	1	2	3
0	0	1	2	5
1	1	0	1	4
2	2	1	0	3
3	1	2	3	0

表 5.1: D の場合の各ステップ数

今回の評価で実際に測定の対象としたのは各枝のうち最も小さいものと最も大きいものを選んでいった場合の子孫である 2^{n-2} 通りである。状態数 4 の場合は A、 C、 D、 F が測定の対象となる。

5.1.2 測定の手順と結果

1. 状態数 $n(n = 1, 2, \cdots, 20)$ の状態の木とそれを作る時にできる最小のヒストリを 5.1.1節で述べた組合せについて作る。

2. ある状態から別のある状態に移動するまでのステップ数を

   (a) 木構造モデル

   (b) ヒストリモデルにおける最小値

   (c) ヒストリモデルにおける平均値

   (d) ヒストリモデルにおける最大値

   について調べる。なお、ヒストリモデルでは同じ状態が複数存在するため、状態 α から状態 β までのステップ数は複数の状態 α の中から一つを選び、そこから複数ある状態 β までの距離で最小のものとする。

   そのようにして複数の状態 α からの状態 β までのステップ数を求め、それらの最小値、平均値、最大値が 2b、 2c、 2dである。

3. それらのそれぞれについて測定したものの平均をとる。

例えば図 5.1で、 D について上記 2を測定したものが表 5.1である。

このようにしてできた各表の値の平均値を求め、さらにそれらを A、 C、 D、 F について平均することによって表 5.2を得る。

同様にして状態数 n(n=1,2, ⋯ 20) の場合の平均ステップ数を求めたのが図 5.2である。なお、これらの測定には付録 Aに示したプログラムを使用した。

木構造	ヒストリ最小	ヒストリ平均	ヒストリ最大
1.1	1.3	1.4	1.6

表 5.2: 状態数 4 の場合のステップ数の平均値

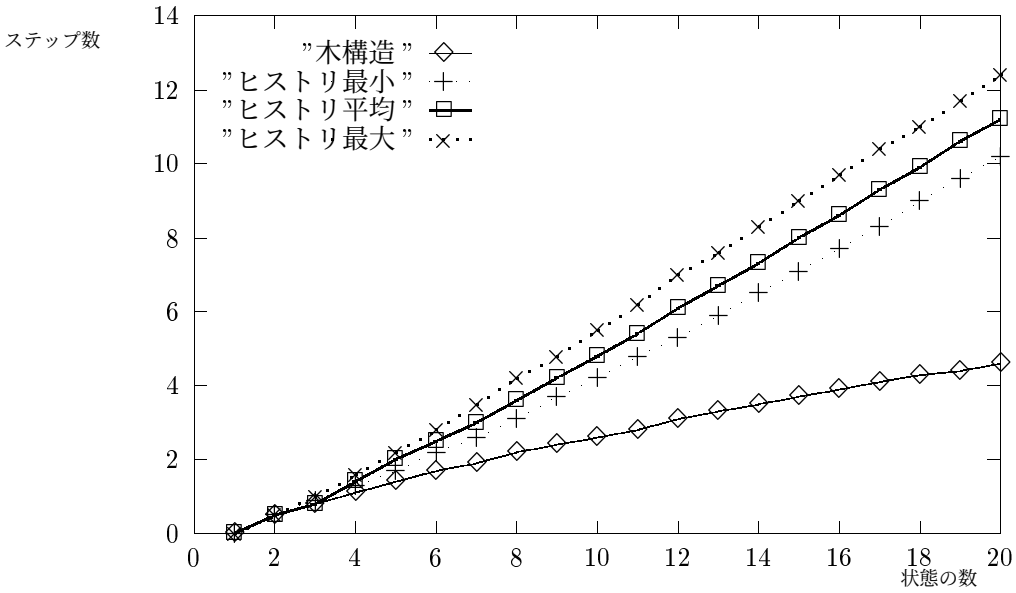


図 5.2: ステップ数の平均値

状態の数

5.2 インタフェース

ここでは3.3節で述べた 8 つのインタフェースのうち、 cross、 list、 treeGraph、 linear、 linearGraph、 allGraph の 6 つについての評価を行う。

まず、このシステムをはじめて使う 4 人の被験者に次のようなことをおこなってもらった。

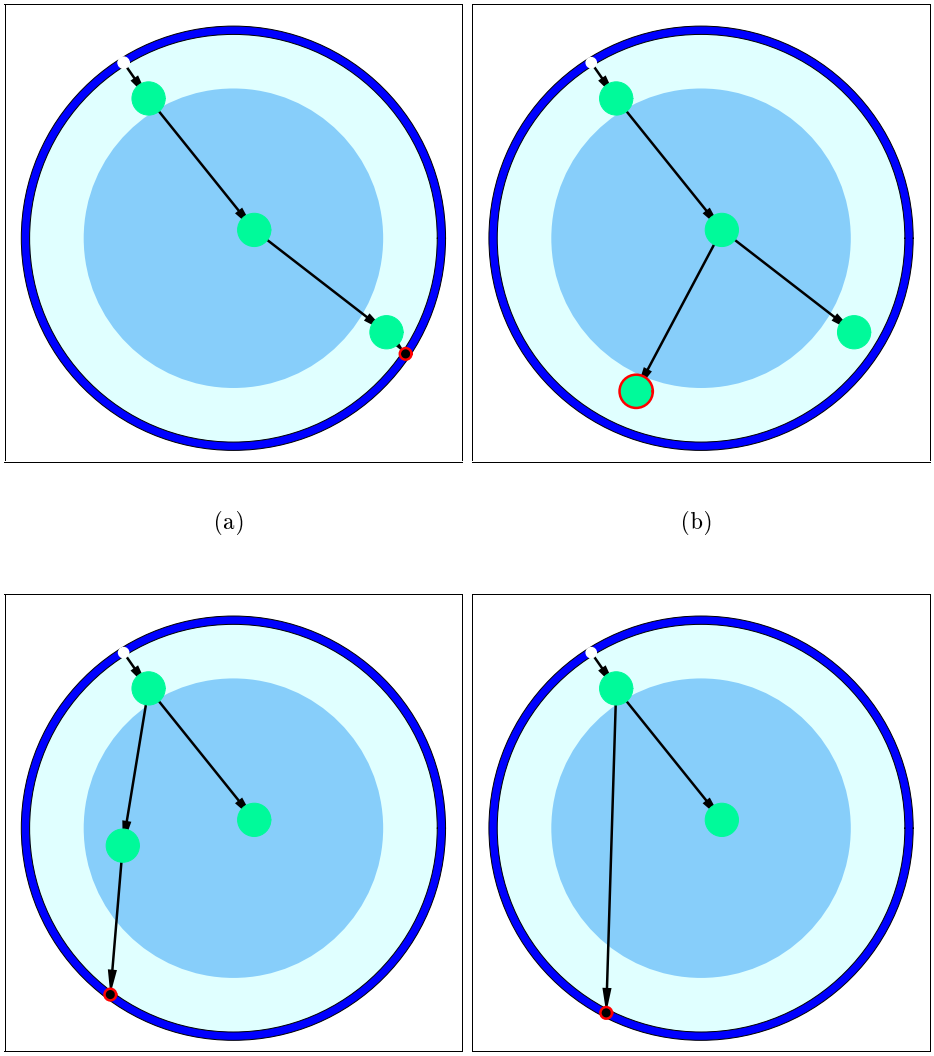
1. ベースである PP について説明を受ける。

2. 上述した 6 つのインタフェースそれぞれについて

   ① 説明を受け、 1 分程度練習をする。

   ② 図 5.3に示す 4 つの状態を (a)、 (b)、 (c)、 (d) の順に作る。ただし、「消去」コマンドは使わず、各状態の中に表示されているアイテムを消去する場合には必ず「undo」コマンドを用いるものとする。

   ③ 図 5.3の (d) の状態から (a) の状態まで undo する。



(a) (b)

(c) (d)

図 5.3: 作る状態

このうち、③にかかった

- 合計時間 (t_{sum})
- 実際のステップ数 (s_{act})

を測定し、それらと理論上の最小ステップ数 (s_{min}) とを用いて次の値を求めた。

- error 率 (e)
- 1 ステップ当たりの時間 (t_{one})

ここで、

$$e = \frac{s_{act} - s_{min}}{s_{act}}$$

$$t_{one} = \frac{t_{sum}}{s_{act}}$$

である。

error 率は「どれだけ少ない失敗で目的の状態まで戻れるか」の指標となり、1 ステップ当たりの時間は「どれだけ悩むことなく作業できるか」の指標となる。

しかし 1 ステップ当たりの時間が短くても error 率が高くては意味がないし、その逆も言える。そこで、これら両方から決まる値を求める。ここではその値 v を次のように定義する。

$$v = \frac{t_{one}}{1 - e}$$

1 ステップ当たりの時間が長くてもエラーが多くても v の値は大きくなる。これに t_{one} と e の式を入れると、実は

$$v = \frac{t_{sum}}{s_{min}}$$

となる。

実際の測定を行い、平均を求めた結果は表 5.3のようになった。

	s_{min}[step]	t_{sum}[sec]	s_{act}[step]	t_{one}[sec/step]	e	v[sec/step]
cross	4	77.6	12.8	6.1	0.69	19.4
treeGraph	4	111.3	10.3	10.9	0.61	27.8
linear	7	26.8	8.0	3.3	0.13	3.8
linearGraph	7	28.3	7.0	4.0	0	4.0
list	3	72.8	9.0	8.1	0.67	24.25
allGraph	1	16	1.0	16.0	0	16

表 5.3: 各インタフェースの測定の結果

この結果及び被験者の意見から、木構造を用いたインタフェースはかなり使いにくいということが言える。これには次のような理由があげられる。

- undo するときに頭の中に木構造を描いていなければならない。
- 木構造の全体が見えていないのでどの方向に進めるのか分からない。

とくに、ミニチュアを用いている treeGraph では実は目的の状態が現在の状態の子供であるのに、最も若い子のみが表示されているために被験者がその状態に移動しようとしなかったということがよくあった。

したがって木構造全体を表示するようなインタフェースを作ればこれらの問題は解決するものと思われる。

一方、ヒストリを用いたインタフェースで v が小さいのは、何も考えなくてもとにかく undo していれば必ず目的の状態に戻れるということで e、t_{one} ともに小さくなっていることが原因である。

ミニチュアを用いている linearGraph がミニチュアを用いていない linear よりも t_{one} が大きくなってしまうのは、ミニチュアを見て確認しながら作業を行ってしまうということが原因であると考えられる。これは何も考えなくてもいいというヒストリを用いたインタフェースの利点をかえって無駄にしてしまっている。

allGraph で t_{one} が大きくなるのはたくさん並んだ状態のなかから目的の状態を探すのに時間がかかっているためと思われる。今回の測定では作られる状態の数は 9 個である。実際に作業を行う場合はもっと状態の数が増えるので探すときにはもっと時間がかかるであろう。

第 6 章

まとめ

本研究では木構造を用いた undo の提案とそれを用いたビジュアルプログラミングシステム PP 上での undo 機能について述べた。 PP の undo 機能のインタフェースではミニチュアも用いた。

木構造を用いた undo は従来の方法に比べて理論的には少ないステップ数で戻ることができるが、構造が複雑であるためにわかりやすいインタフェースを用いないと従来の方法よりもはるかに多くの時間とステップ数がかかってしまうことがわかった。

ミニチュアは従来のボタンを用いたものと比較してそれ自体が持っている情報が多いということが特徴であるが、そのことがかえってユーザは混乱させる場合があるので注意が必要であるということもわかった。

これらのことをふまえた上でより使いやすいインタフェースを作らなければならない。

謝辞

本研究を進めるにあたり終始ご指導下さった田中二郎先生に感謝する。光延秀樹君、南雲淳君、 Ali Jauhar さん、寺茂夫君は忙しい中インタフェースの評価を行ってくれた。また、田中研究室のみなさんからは数々の貴重な助言を頂いた。ここで感謝の意を表する。

参考文献

[1] Donald A.Norman 著, 野島久雄訳. 誰のためのデザイン？ 新曜社, 1990.

[2] 海保博之, 加藤隆. 人に優しい コンピュータ画面設計ユーザ・インタフェース設計への認知心理学的アプローチ. 日経 BP 社, 1992.

[3] BenShneiderman 著, 東基衛, 井関治監訳. ユーザー・インタフェースの設計 使いやすい対話型システムへの指針. 日経 BP 社, 1987.

[4] RichardStallman 著, 竹内郁雄, 天海良治監訳. GNU Emacs マニュアル. 共立出版, 1988.

[5] Apple Computer,Inc. Human Interface Guidelines:The Apple Desktop Interface(日本語版). アジソン・ウエスレイ・パブリシャーズ・ジャパン, 1990.

[6] William Chia-Wei Cheng. URL:http://bourbon.cs.ucla.edu:8001/tgif/.

[7] 後藤和貴. ビジュアルプログラミングシステムにおける図形入力の考察. 筑波大学第三学群情報学類卒業論文, 1995.

[8] 田中二郎, 後藤和貴, 馬場昭宏. ビジュアルプログラミングシステムにおける入力法の効率化. 日本ソフトウェア科学会第 12 回大会論文集, pp. 165–168, 1995.

[9] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley Publishing Company, 1994.

付録 A

評価に使用したプログラム

```
/***** eval.c *****/

/*************************************************************************
   This program makes state_trees that consist of certain number of
  elements.  The number is given by user when this program starts.Then
  measures the values described in section 5.1.2.
   If you need the values of each tree,compile with -DDEBUG option.

   WARNING:This program doesn't check if the value entered is leagal.
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define MAX_STATES 20           /* max value of 'number_of_states' */
#define MAX_HISTORY_LENGTH 1000 /* max length of the history */

#define OK 1
#define EOL -1

#ifdef DEBUG
#       define Dbg(a)   a
#else
#       define Dbg(a)
#endif

typedef struct tree {  /* See section 4.1 */
  int up;
  int down;
  int left;
  int right;
} TREE;

typedef struct history {
  int list[MAX_HISTORY_LENGTH];
  int length;                   /* current length of this history */
} HISTORY;

typedef struct table {
  TREE tree[MAX_STATES]; /*    This and */
  HISTORY history;       /* this are equivalent. */
} TABLE;
```

```
typedef struct distance {  /* See section 5.1.2 */
  double tree;
  double history_max;
  double history_avr;
  double history_min;
} DISTANCE;

void make_initial_table(TABLE *table);
/*
 * Function: make initial table,that is,table whose element is only
 *          state '0'
 * Parameters:
 *      TABLE *table: pointer to the table which you want to initialize
 * Return value: none
 */

int make_tree_and_measure(int number_of_states,int parent,
                          int child,TABLE table);
/*
 * Function: make trees and histories recursively and measure the
 *          values.
 * Parameters:
 *      int number_of_states: number of states(nodes) of the trees and
 *                            histories to make and measure
 *      int parent,: New state represented by 'child' is added to the
 *      int child, :tree and history represented by 'table' as a child
 *      TABLE table:of the state represented by 'parent'.
 * Return value: OK
 * Side effects: value of 'number_of_trees' and 'dist_avr' will be changed.
 */

void copy_table(TABLE from,TABLE *to);
/*
 * Function: copy the contents of a table to another
 * Parameters:
 *      TABLE from: the table to be copied from
 *      TABLE to  : pointer to the table to copy to
 * Return value: none
 */

void add_new(TABLE *table,int parent,int child);
/*
 * Function: add new state to the table(tree and history)
 * Parameters:
 *      int parent,: New state represented by 'child' is added to the
 *      int child, :tree and history represented by 'table' as a child
 *      TABLE table:of the state represented by 'parent'.
 * Return value: OK
 */

void intcpy(int *from,int *to);
/*
 * Function: copy an array of integers to another array
 * Parameters:
 *      int *from: pointer to the array to be copied from
 *      int *to  : pointer to the array to copy to
 * Return value: none
 */

void measure(int number_of_states,TABLE table);
/*
```

```
 * Function: measure values described in section 5.1.2
 * Parameters:
 *      int number_of_states: number of states(nodes) of the trees and
 *                            histories to make and measure
 *      TABLE table: table to be measured
 * Return value: none
 */

void print_history(HISTORY history);
/*
 * Function: print the given history
 * Parameters:
 *      HISTORY history: history to print
 * Return value: none
 */

int tree_len(TABLE table,int From,int To);
/*
 * Function: measure the value(See Parameters below) of tree
 * Parameters:
 *      TABLE table,: The length of the path from the node 'from' to
 *      int From,   :the node 'To' in the tree 'table' is measured.
 *      int To      :
 * Return value: measured value
 */

int real_to_path_len(TABLE table,int From,int To);
/*
 * Function: measure the length of the path(See Parameters below).The
 *          path is NOT "logical" one but "real" one.
 * Parameters:
 *      TABLE table,: The length from the node 'To' to the node 'From'
 *      int From,   :in the tree 'table' is measured.
 *      int To      :
 * Return value: measured value
 * WARNING: 'To' MUST be ancestor of 'From'
 */

int history_distance(int pointer,int to,HISTORY history);
/*
 * Function: measure the shortest length(See Parameters below) in the
 *          history.
 * Parameters:
 *      int pointer,   : The length from 'pointer'th element in the
 *      int to,        :history 'history' to the nearest element
 *      HISTORY history:whose value is state 'to' is measured.
 * Return value: measured value
 */


DISTANCE distance[MAX_STATES][MAX_STATES]; /* To tell the truth,this is
                                              not necessary to be an array */
DISTANCE dist_avr; /* averages of the values(See section 5.1.2.) */
int number_of_trees; /* number of the trees made */


void main() {
  TABLE state_table;    /* table that represents a tree and a history */
  int number_of_states; /* number of states(nodes) of the trees and
                           histories to make and measure           */
  char buffer[5];       /* used in gets */
```

```
  int divider;          /* See comments below. */

  /* get 'number_of_states' */
  printf("number_of_states(1-%d)=",MAX_STATES);
  number_of_states=atoi(gets(buffer));
  printf("\n");

  /* initialize the table and values */
  make_initial_table(&state_table);
  dist_avr.tree=0;
  dist_avr.history_max=0;
  dist_avr.history_avr=0;
  dist_avr.history_min=0;
  number_of_trees=0;

  /* make trees and histories and measure the values described in
     section 5.1.2 */
  make_tree_and_measure(number_of_states-1,0,1,state_table);

  /* Values measured in 'make_tree_and_measure' is the sum of all the
    length of each 'from' and 'to' pairs. */

  /* get the value of 'divider' */
  divider=number_of_trees*number_of_states*number_of_states;

  /* get the values by dividing by 'divider' */
  dist_avr.tree/=divider;
  dist_avr.history_max/=divider;
  dist_avr.history_avr/=divider;
  dist_avr.history_min/=divider;

  /* print the values */
  printf("tree=%6.2f\n",dist_avr.tree);
  printf("history max=%6.2f\n",dist_avr.history_max);
  printf("history minimum=%6.2f\n",dist_avr.history_min);
  printf("history average=%6.2f\n\n",dist_avr.history_avr);
}


void make_initial_table(TABLE *table) {

  int i; /* used in the loop */

  /* initialize the tree */
  for (i=0;i<MAX_STATES;i++) {
    table->tree[i].up=-1;
    table->tree[i].down=-1;
    table->tree[i].left=-1;
    table->tree[i].right=-1;
  }

  /* initialize the history */
  table->history.list[0]=0;
  table->history.list[1]=EOL;
  table->history.length=1;
}

int make_tree_and_measure(int number_of_states,int parent,
                          int child,TABLE table) {

  TABLE state_table; /* used to pass the table to next tree */
```

```
  /* copy the given table to reserve it */
  copy_table(table,&state_table);

  /* add new state to the table */
  add_new(&state_table,parent,child);

  /* In this case,the tree with 'number_of_states' of element is
     completely made */
  if (child==number_of_states) {
    measure(number_of_states,state_table);
    return OK;
  }

  /* make new two trees by adding new state('child+1') to state '0'
    and to state 'child'.  */
  make_tree_and_measure(number_of_states,0,child+1,state_table);
  make_tree_and_measure(number_of_states,child,child+1,state_table); }


void copy_table(TABLE from,TABLE *to) {

  int i; /* used in the loop */

  /* copy tree */
  for (i=0;i<MAX_STATES;i++) {
    to->tree[i].up=from.tree[i].up;
    to->tree[i].down=from.tree[i].down;
    to->tree[i].left=from.tree[i].left;
    to->tree[i].right=from.tree[i].right;
  }

  /* copy history */
  intcpy(from.history.list,to->history.list);
  to->history.length=from.history.length;
}


void add_new(TABLE *table,int parent,int child) {

  int ptr;           /* used to indicate the point in the history */
  int older_brother;

  /* add new state to the tree */
  if (table->tree[parent].down!=-1) {
    older_brother=table->tree[parent].down;
    table->tree[older_brother].left=child;
    table->tree[child].right=older_brother;
  }
  table->tree[parent].down=child;
  table->tree[child].up=parent;

  /* add new state to the history */
  ptr=table->history.length-1;
  while (table->history.list[ptr]!=parent) {
    ptr--;
    table->history.list[table->history.length]=table->history.list[ptr];
    table->history.length++;
  }
  table->history.list[table->history.length]=child;
  table->history.length++;
```

```
  table->history.list[table->history.length]=EOL;
}


void measure(int number_of_states,TABLE table) {

  int i;           /* used in the loop */
  int from;        /* measure values from the node 'from'*/
  int to;          /*to the node 'to' */
  double hist_max; /* max length of history */
  double hist_min; /* minimum length of history */
  double hist_sum; /* sum of lengths of history */
  int num;         /* number of state 'from' in the history */
  int ptr;         /* used to indicate the print in the history */
  int dist;

  /* In DEBUG mode,pointers of tree and the history are printed.This
     means you can know how the tree and the history are.*/
  Dbg(for (i=0;i<=number_of_states;i++) {
    printf("tree[%d]:up=%3d,down=%3d,left=%3d,right=%3d\n",i,
           table.tree[i].up,table.tree[i].down,
           table.tree[i].left,table.tree[i].right);
  }
  print_history(table.history);
  printf("\n");
      )

  for (from=0;from<=number_of_states;from++) {
    for (to=0;to<=number_of_states;to++) {

      /* get the distance in the tree */
      distance[from][to].tree=tree_len(table,from,to);

      /* get distances in the history */
      /** initialize **/
      hist_max=0;
      hist_min=table.history.length-1;
      hist_sum=0;
      num=0;

      /** A history has some same states.In this part,get the shortest
         distance from all 'from's to 'to' **/

      for (ptr=0;ptr<table.history.length;ptr++) {
        if (table.history.list[ptr]==from) {
          dist=history_distance(ptr,to,table.history);
          if (dist>hist_max) {
            hist_max=dist;
          }
          if (dist<hist_min) {
            hist_min=dist;
          }
          hist_sum+=dist;
          num++;
        }
      }

      /** set values measured above to distance_table **/
      distance[from][to].history_max=hist_max;
      distance[from][to].history_avr=hist_sum/num;
      distance[from][to].history_min=hist_min;
```

```
      /* add values.These values are to be divided in main function. */
      dist_avr.tree+=distance[from][to].tree;
      dist_avr.history_max+=distance[from][to].history_max;
      dist_avr.history_avr+=distance[from][to].history_avr;
      dist_avr.history_min+=distance[from][to].history_min;

      /*In DEBUG mode,print distance from 'from' to 'to'*/
      Dbg(printf("%5.1f/",distance[from][to].tree);)
      Dbg(printf("%5.1f/",distance[from][to].history_max);)
      Dbg(printf("%5.1f/",distance[from][to].history_avr);)
      Dbg(printf("%5.1f  ",distance[from][to].history_min);)
    }
    Dbg(printf("\n");)
  }
  Dbg(printf("\n");)

  /* increment 'number_of_trees' measured */
  number_of_trees++;
}


int history_distance(int pointer,int to,HISTORY history) {

  int ptr;   /* used in the loop */
  int left;  /* shortest path in the left from 'pointer'*/
  int right; /* shortest path in the right from 'pointer' */

  /* search 'to' in the left from 'pointer' in the history */
  left=history.length;
  for (ptr=pointer;ptr>=0;ptr--) {
    if (history.list[ptr]==to) {
      left=pointer-ptr;
      break;
    }
  }

  /* search 'to' in the right from 'pointer' in the history */
  right=history.length;
  for (ptr=pointer+1;ptr<history.length;ptr++) {
    if (history.list[ptr]==to) {
      right=ptr-pointer;
      break;
    }
  }

  /* return shortest */
  if (left<right) {
    return left;
  } else {
    return right;
  }
}


void intcpy(int *from,int *to) {

  while (*from!=EOL) {
    *to=*from;
    to++;
    from++;
```

```
  }
  *to=EOL;
}


void print_history(HISTORY history) {

  int i;  /* used in the loop */

  printf("history= ");
  for (i=0;i<history.length;i++) {
    printf("%3d",history.list[i]);
  }
  printf("\n");
}

int tree_len(TABLE table,int From,int To) {

  int from_path[MAX_STATES]; /* path to the root of the tree from 'From' */
  int to_path[MAX_STATES];   /* path to the root of the tree from 'To' */
  int from_path_len;         /* length of 'from_path' */
  int to_path_len;           /* length of 'to_path' */
  int from;
  int to;
  int f;
  int t;
  int found;

  /* make from_path */
  from=From;
  from_path_len=0;
  while (from!=-1) {
    from_path[from_path_len]=from;
    from=table.tree[from].up;
    from_path_len++;
  }

  /* make to_path */
  to=To;
  to_path_len=0;
  while (to!=-1) {
    to_path[to_path_len]=to;
    to=table.tree[to].up;
    to_path_len++;
  }

  /* find common ancestor of 'from' and 'to' */
  found=0;
  for (f=0;f<from_path_len;f++) {
    for (t=0;t<to_path_len;t++) {
      if (from_path[f]==to_path[t]) {
        found=1;
        break;
      }
    }
    if (found==1) {
      break;
    }
  }
  from_path_len=f-1;
  to_path_len=t-1;
```

```
  /* In this case 'to' is ancestor of 'from' */
  if (to_path_len==-1) {
    return from_path_len+1;
  }

  /* In this case 'from' is ancestor of 'to' */
  if (from_path_len==-1) {
    return real_to_path_len(table,To,to_path[to_path_len+1]);
  }

  /* add length between ancestor of 'to' and that of 'from' that are brothers
     to 'from_path_len'.*/
  if (from_path[from_path_len]>to_path[to_path_len]) {
    from=to_path[to_path_len];
    to=from_path[from_path_len];
  } else {
    from=from_path[from_path_len];
    to=to_path[to_path_len];
  }
  while (from!=to) {
    from=table.tree[from].left;
    from_path_len++;
  }
  return from_path_len+real_to_path_len(table,To,to_path[to_path_len]);
}


int real_to_path_len(TABLE table,int From,int To) {

  int from;
  int length;

  length=0;
  from=From;
  while (from!=To) {
    /* first,move left */
    while (table.tree[from].left!=-1) {
      from=table.tree[from].left;
      length++;
    }
    /* then move up */
    from=table.tree[from].up;
    length++;
  }
  return length;
}
```