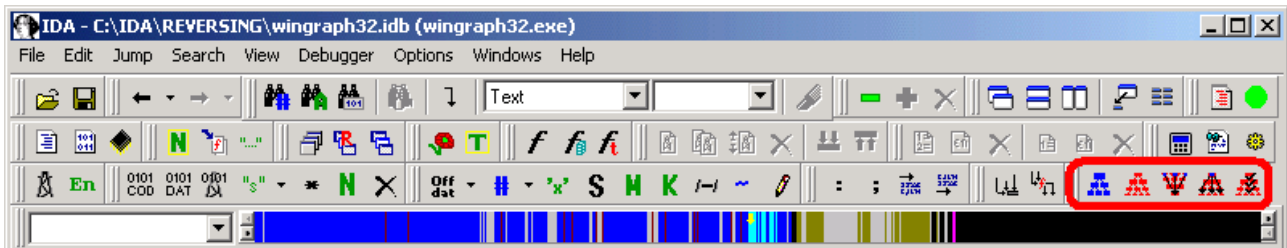


Graphing with IDA Pro. © DataRescue 2005

Wingraph32, a partial port of the VCG graphing library, is available since IDA Pro 4.17. IDA is able to produce standard GDL graphs which are then passed to Wingraph32 for drawing. The graphing commands are available in the graph toolbar.



Let's look more in detail at some of these possibilities.

Flow charts.

Observing a function's code flow on a graph usually gives a better global view of a function's structure than the one we'd get by browsing. The *Flow Chart* command draws such graphs.

The screenshot displays the IDA Pro interface with the assembly code for the `_strncpy` function on the left and its flow graph in the WinGraph32 window on the right. A red circle highlights the 'Graphs' menu icon, and a red arrow points to the WinGraph32 window.

```
0049AF58 ; char *_cdecl strncpy(char *dest,const char *src,size_t maxlen)
0049AF58 _strncpy proc near ; CODE XREF: sub_429BBC+14C↑p sub_429F68+1F↑p
0049AF58 dest= dword ptr 8
0049AF58 src= dword ptr 0Ch
0049AF58 maxlen= dword ptr 10h
0049AF58 push ebp
0049AF59 mov ebp, esp
0049AF5B push ebx
0049AF5C push esi
0049AF5D push edi
0049AF5E mov edi, [ebp+src]
0049AF61 mov esi, [ebp+dest]
0049AF64 push edi ; s
0049AF65 call _strlen
0049AF6A pop ecx
0049AF6B mov ebx, eax
0049AF6D cmp ebx, [ebp+maxlen]
0049AF70 jbe short enough_space
0049AF72 not_enough_space:
0049AF72 mov eax, [ebp+maxlen]
0049AF75 push eax ; n
0049AF76 push edi ; src
0049AF77 push esi ; dest
0049AF78 call _memcpy
0049AF7D add esp, 0Ch
0049AF80 jmp short end
0049AF82 ;
0049AF82 enough_space: ; CODE XREF:
0049AF82 push ebx ; n
0049AF83 push edi ; src
0049AF84 push esi ; dest
0049AF85 call _memcpy
```

The flow graph in WinGraph32 shows the control flow of the function:

- Initial Block:** `_strncpy:` pushes `ebp`, `esp`, `ebx`, `esi`, and `edi`. It then moves `edi` to `[ebp+src]`, `esi` to `[ebp+dest]`, and pushes `edi` with value `s`. It calls `_strlen`, pops `ecx`, moves `ebx` to `eax`, and compares `ebx` with `[ebp+maxlen]`. A `jbe` instruction branches to `enough_space` if true, and `false` otherwise.
- not_enough_space:** Moves `eax` to `[ebp+maxlen]`, pushes `eax` (value `n`), `edi` (value `src`), and `esi` (value `dest`). It calls `_memcpy`, adds `esp` by `0Ch`, and jumps to `end`.
- enough_space:** Pushes `ebx` (value `n`), `edi` (value `src`), and `esi` (value `dest`). It calls `_memcpy`, adds `esp` by `0Ch`, moves `edi` to `[ebp+maxlen]`, subtracts `ebx` from `ebx`, adds `esi` to `ebx`, pushes `edi` (value `n`), `0` (value `c`), and `ebx` (value `s`). It calls `_memset` and adds `esp` by `0Ch`.
- end:** Moves `eax` to `esi`, pops `edi`, `esi`, `ebx`, and `ebp`, and returns.

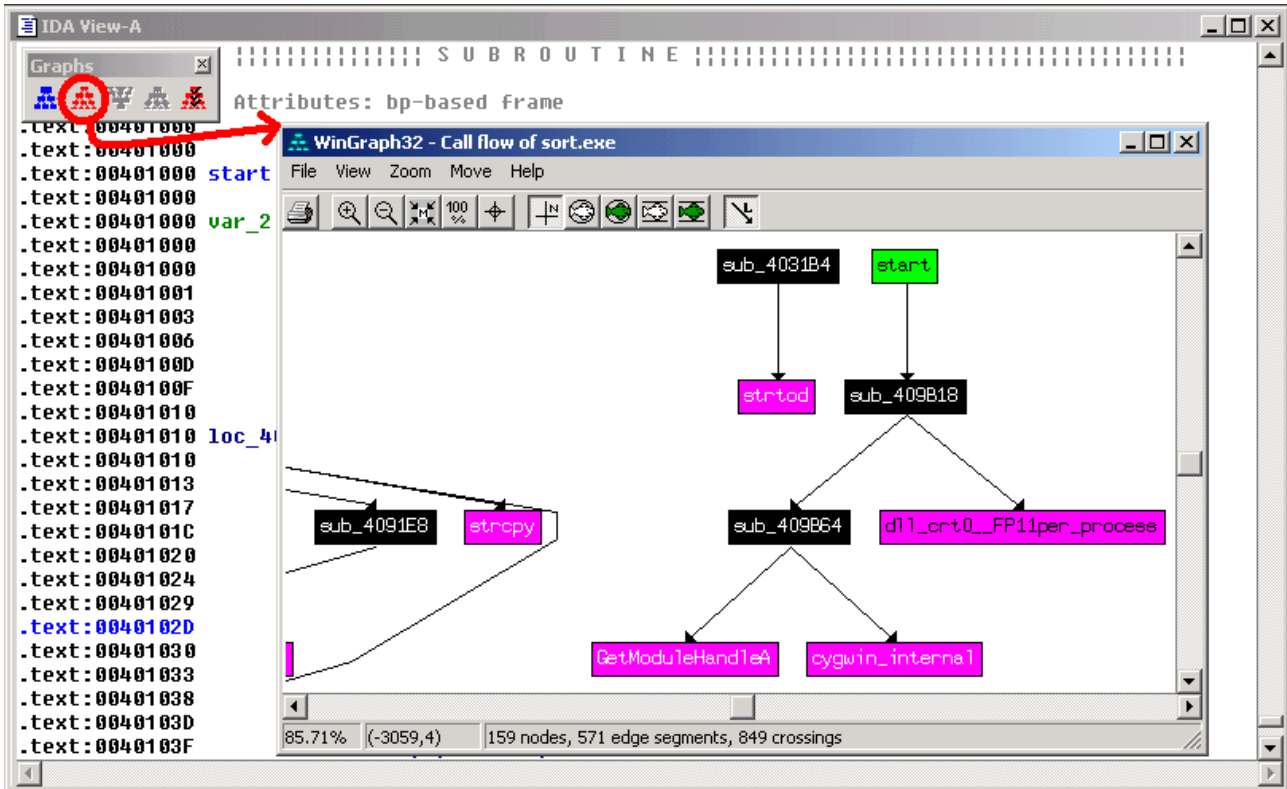
The status bar at the bottom indicates: 55.56% (0,0) 4 nodes, 8 edge segments, 0 cro:

If you want to graph several functions at the same time, or to graph only a portion of a larger function, just select the area of interest using our standard selection keys.

The screenshot displays the IDA Pro interface with the assembly view on the left and the WinGraph32 control flow graph on the right. The assembly code shows a function `sub_402608` with a switch statement starting at address `00402614`. The switch statement compares `eax` against the value `7` and branches based on the result. The control flow graph visualizes this logic, showing a root node at `00402614` that branches into a `true` path and a `false` path. The `true` path leads to a `default` node, while the `false` path leads to a `use_jump_table` node. This node then branches into seven cases: `eax_7`, `eax_5`, `eax_0`, `default1`, `eax_1`, `eax_6`, and `eax_4`. The assembly view shows the corresponding instructions for each case, including `add`, `cmp`, `ja`, `jmp`, and `test` instructions. The `test` instruction at `00402647` is highlighted, indicating the current selection.

Function calls.

Usually, analyzing dependences between functions is necessary to understand how a particular application works. IDA offers us a command to graph all existing dependences (cross references) between program functions.

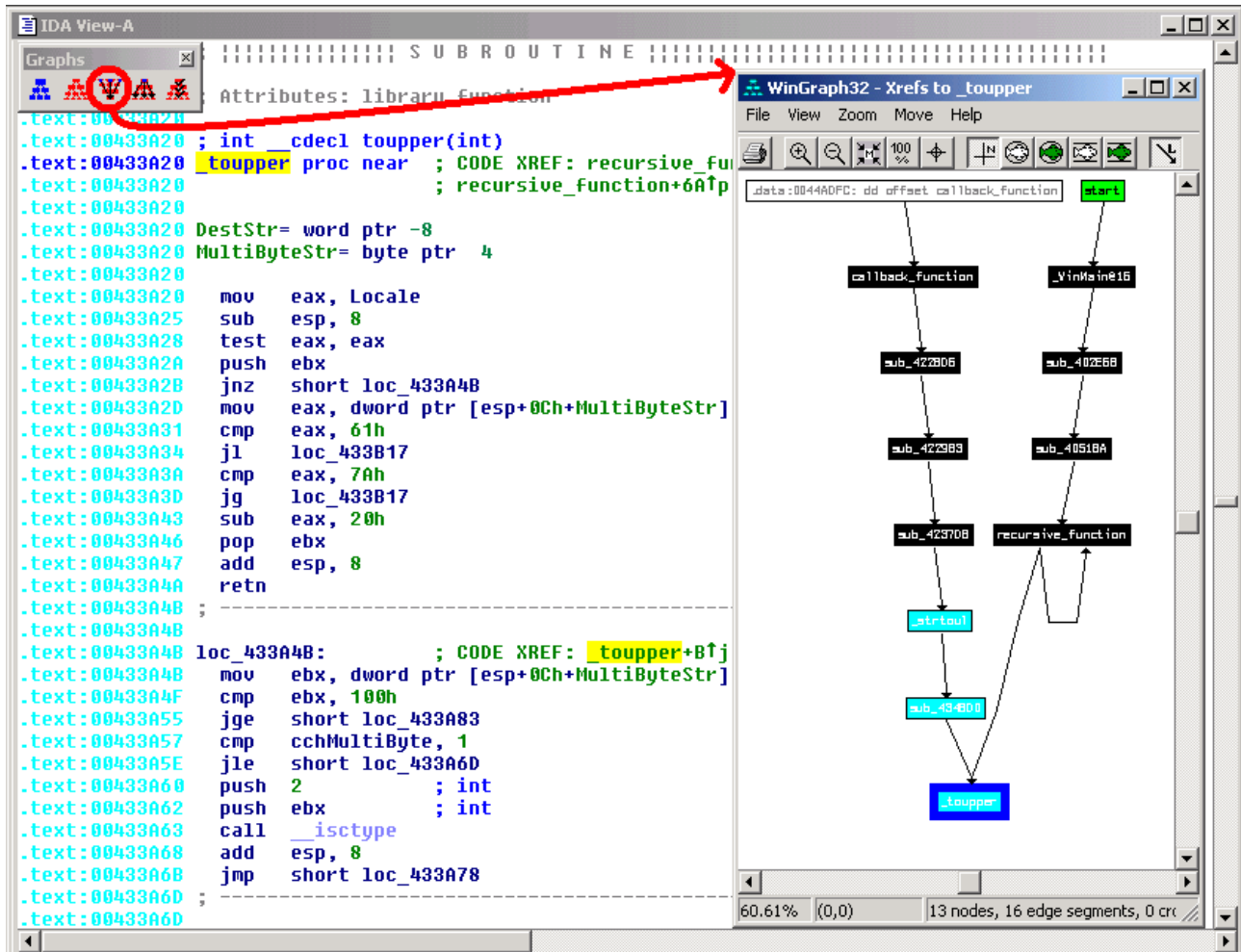


This command is mainly interesting for small programs, because the graph quickly becomes extremely complex in the case of a program containing a large number of functions. Notice that the graph's colours depend on functions/addresses attributes (externals functions, libraries functions, entry point, ...). The colour scheme is coherent with the one of the disassembly view.

Function cross references.

Xrefs to and *Xrefs from* allow you to focus on cross references to and from a particular function.

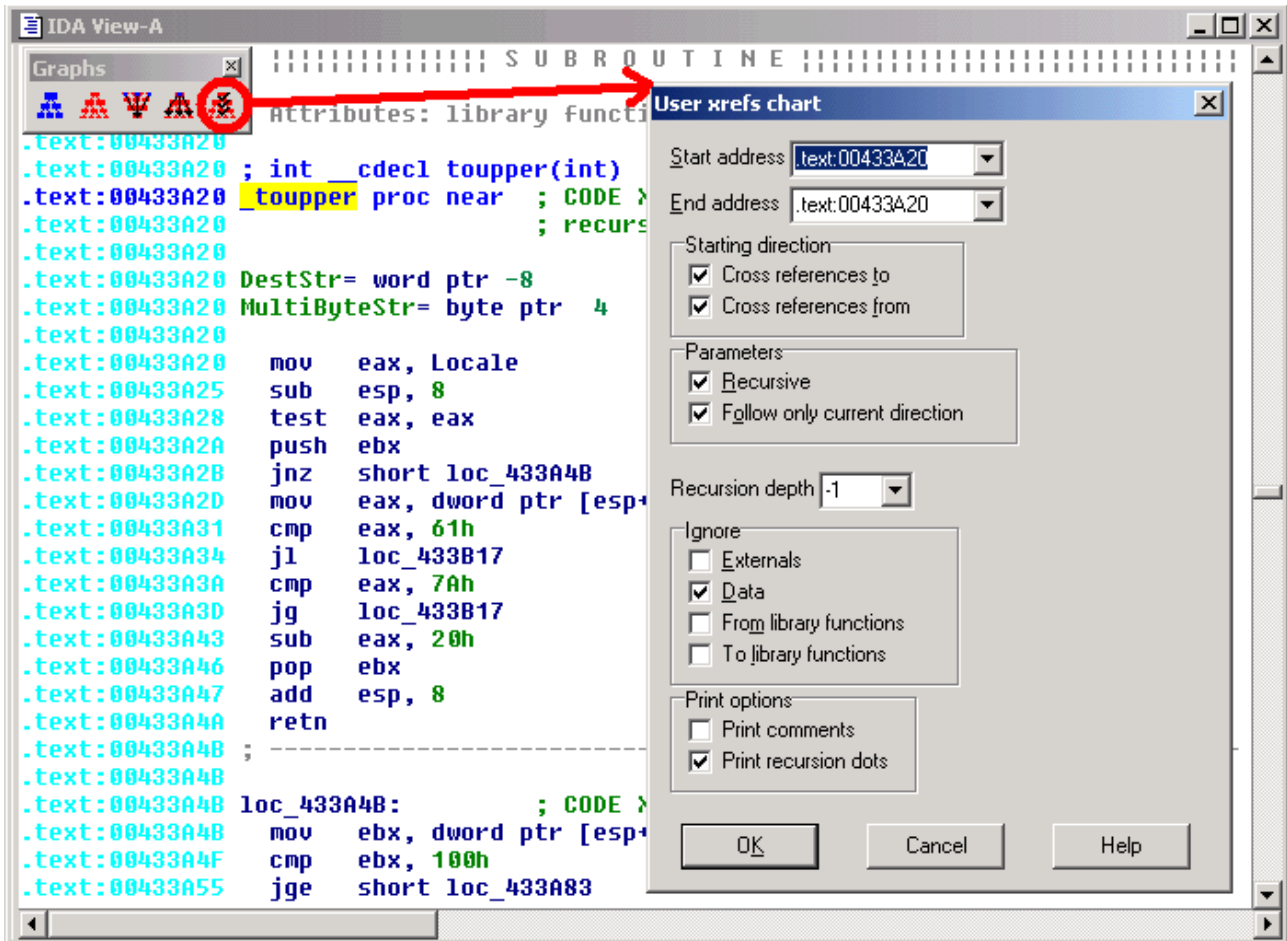
The *Xrefs to* command graphs data and code cross references leading to a given function. Let's have a look at the *toupper()* function, from the C Standard Library, that translates characters to upper case.



At the top of the graph, we notice the *start()* function: it calls the *WinMain()* function. On the left, we observe a function pointer to the *callback_function()*. On the bottom right, we can observe the *recursive_function()*, calling itself. We also remark that the *strtol()* function (from the C Standard Library, who converts a string to an unsigned long) indirectly depends itself on the *toupper()* function. Finally, notice that the selected function is always surrounded by a blue frame.

User defined cross references chart.

IDA offers advanced graphing functions for more sophisticated users. Let's see how we can use them in practice.



Let's take some practical cases requiring the use of these options.

Data cross references.

Remember that the *Xrefs from* command only **draws code cross references**. But sometimes, it could be useful to also follow data cross references. Let's apply this to observe a pointer to an array of function pointers, by unselecting the *Cross references to* and *Ignore Data* options.

The screenshot displays the IDA Pro interface. The main window shows assembly code with the following lines:

```
.data:00449797 db 0 ;  
.data:00449798 function_pointers_array dd offset func1, offset func2, offset func3, 0  
.data:00449798 ; DATA XREF: .data:004497A8↓  
.data:004497A8 pointer_to_the_array dd offset function_pointers_array  
.data:004497A8 ; DATA XREF: sub_414AF8+88↑  
.data:004497AC db 0 ;
```

The 'User xrefs chart' dialog box is open, showing the following settings:

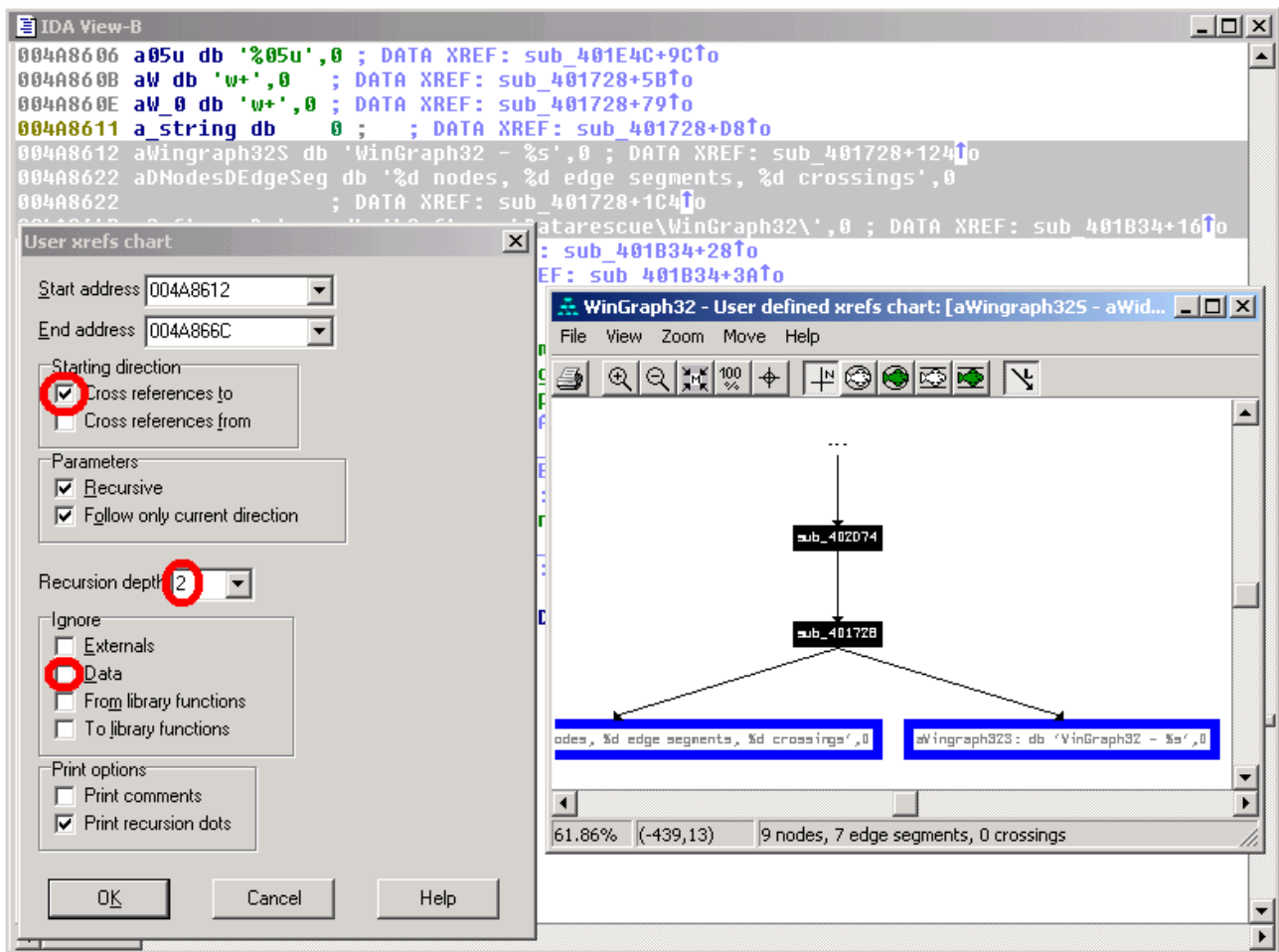
- Start address: .data:004497A8
- End address: .data:004497A8
- Starting direction:
 - Cross references to
 - Cross references from
- Parameters:
 - Recursive
 - Follow only current direction
- Recursion depth: -1
- Ignore:
 - Externals
 - Data
 - From library functions
 - To library functions
- Print options:
 - Print comments
 - Print recursion dots

The 'WinGraph32' window displays a graph with the following structure:

- Root node: pointer_to_the_array: dd offset function_pointers_array
- Child node: function_pointers_array: dd offset func1, offset func2, offset func3, 0
- Leaf nodes: func1, func3, func2

Red circles highlight the 'Cross references to' checkbox in the dialog and the 'OK' button. A red arrow points from the 'OK' button to the graph window.

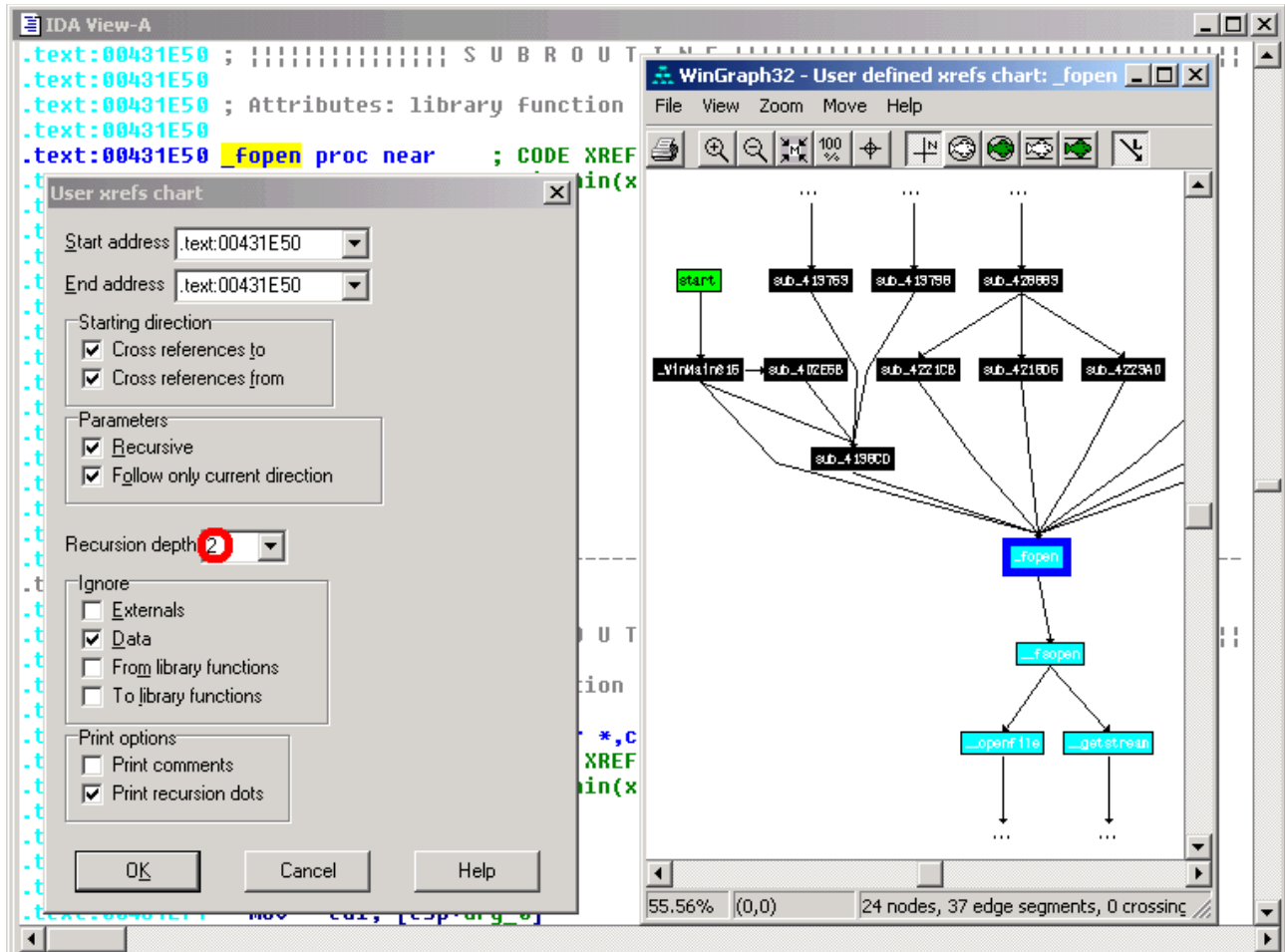
Yet another possibility is to show references to global data. We can show references to global data from a function, or search for functions referencing a set of global data referenced by the cross referenced functions. It is a good idea to specify the maximum recursion depth, to avoid an almost unreadable graph.



Because we left the *Print recursion dots* option checked, some dots appear on the top, indicating that at least one cross reference exists outside of the range specified by the given recursion depth.

Recursion depth.

As we mentioned previously, the *Function calls* command isn't useful if used on a normal program containing a large number of functions. Again, by specifying a *Recursion depth*, we can try to get the same style of graph, but focused on a particular function. Let's try this on the *fopen()* function.

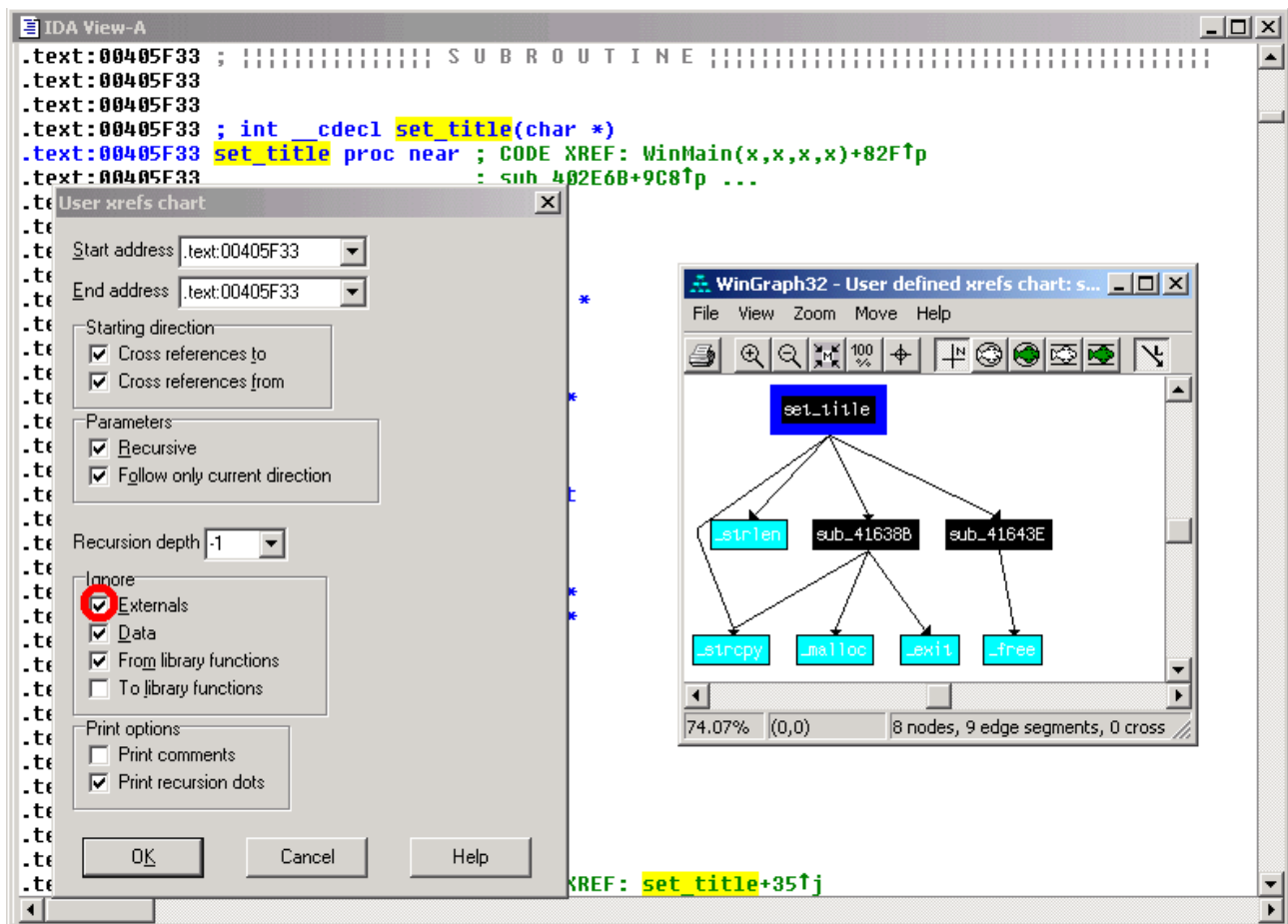


Ignoring specific functions.

IDA's FLIRT (Fast Library Identification and Recognition Technology) identifies standard functions from many libraries. Since these functions are richly documented, we usually aren't too interested by their internals. To hide all those internal cross references and obtain a simpler, more readable graph, we can simply activate the *Ignore From library functions* option.

The screenshot shows the IDA Pro interface. The main window displays assembly code for a subroutine named `set_title` starting at address `00405F33`. The code includes a function declaration and a call to `set_title` near the `WinMain` function. Overlaid on the main window is the 'User xrefs chart' dialog box. In this dialog, the 'Ignore' section has the checkbox 'From library functions' checked, which is highlighted with a red circle. Other options like 'Externals', 'Data', and 'To library functions' are unchecked. The 'Print options' section has 'Print recursion dots' checked. The 'WinGraph32' window shows a graph for the `set_title` function. The graph starts with a root node `set_title` (blue), which branches into five nodes: `SetWindowTextA` (pink), `IsIconic` (pink), `sub_416368` (black), `sub_41643E` (black), and `_str len` (cyan). The `sub_416368` node further branches into five nodes: `_strncpy` (cyan), `_exit` (cyan), `MessageBoxA` (pink), `_malloc` (cyan), and `_free` (cyan). The status bar at the bottom of the graph window shows '68.97% (0,0) 11 nodes, 12 edge segments, 0 crossings'. At the bottom of the IDA window, the text 'XREF: set_title+35↑j' is visible.

However, cross references to external Windows APIs such as `MessageBox()` are always present on our graph. If we also want to hide those cross references, and really focus on the dependence of a function on library functions, we can also activate the *Ignore Externals* option.



Conversely, we could also draw cross references to external Windows APIs, and ignore cross references to library functions. This shows how much a function depends on DLLs.

The screenshot displays the IDA Pro interface with a disassembly window and two xrefs chart windows. The disassembly window shows the following code:

```
.text:00405F33 ; SUBROUTINE  
.text:00405F33  
.text:00405F33  
.text:00405F33 ; int __cdecl set_title(char *)  
.text:00405F33 set_title proc near ; CODE XREF: WinMain(x,x,x,x)+82Ffp  
.text:00405F33 ; sub_402E6B+9C8fp ...
```

The 'User xrefs chart' dialog box is open, showing the following settings:

- Start address: .text:00405F33
- End address: .text:00405F33
- Starting direction:
 - Cross references to
 - Cross references from
- Parameters:
 - Recursive
 - Follow only current direction
- Recursion depth: -1
- Ignore:
 - Externals
 - Data
 - From library functions
 - To library functions
- Print options:
 - Print comments
 - Print recursion dots

The 'WinGraph32 - User defined xrefs chart: set_title' window displays a graph with the following nodes and edges:

- Root node: set_title (blue box)
- Child nodes: SetWindowTextA (pink box), IsIconic (pink box), sub_41643E (black box), sub_416388 (black box)
- Edge from sub_416388 to MessageBoxA (pink box)

The graph statistics at the bottom indicate: 82.19% zoom, (0,0) coordinates, 6 nodes, 5 edge segments, and 0 crossings.

Printing comments.

Let's complete this small tutorial about IDA's graphing features, by presenting a way to graph the dependency of a Windows application on functions from a particular DLL. This can be achieved by creating a graph based on a selection of all external declarations related to this particular DLL. We can also activate the *Print comments* option: it will print comments associated with these external functions. The example below shows how much a program depends on functions exported by *comdlg32.dll*.

The image shows two windows from IDA Pro. On the left is the 'User xrefs chart' dialog box, which is used to configure the graphing process. It includes fields for 'Start address' (00462930) and 'End address' (00462940). Under 'Starting direction', both 'Cross references to' and 'Cross references from' are checked. Under 'Parameters', 'Recursive' and 'Follow only current direction' are checked. The 'Recursion depth' is set to -1. In the 'Ignore' section, 'Externals' is unchecked, while 'Data', 'From library functions', and 'To library functions' are checked. In the 'Print options' section, both 'Print comments' and 'Print recursion dots' are checked. On the right is the 'WinGraph32' window, titled 'User defined xrefs chart: [_imp_ChooseFont...'. It displays a control flow graph starting from a green 'start' node, leading to a black '_WinMain@16' node. This node branches into two black nodes: 'sub_4071C0' and 'sub_402E8B'. 'sub_4071C0' leads to 'sub_4071F5', and 'sub_402E8B' leads to 'sub_40C0FC'. Both 'sub_4071F5' and 'sub_40C0FC' lead to a central black node 'sub_40733D'. From 'sub_40733D', the graph branches into four pink nodes: 'ChooseColorA', 'GetSaveFileNameA', 'ChooseFontA', and 'GetOpenFileNameA'. The 'ChooseColorA' node leads to a blue node 'imp_GetSaveFileNameA' with the comment '; Create a Save common dialog box'. The 'ChooseFontA' node leads to a blue node 'imp_ChooseFontA' with the comment '; Create a Font common dialog box'. The bottom status bar of the graph window shows '66.67% (-215,-3) 15 nodes, 15 edge segments, 0 crossings'. Below the graph window, a portion of the IDA Pro disassembly window is visible, showing assembly code for imports from *comdlg32.dll*.

```
.idata:00462930 ; Imports from comdlg32.dll
.idata:00462930 ;
.idata:00462930 ;
.idata:00462930 extrn __imp_ChooseFontA:dword ; Create a Font common dialog box
.idata:00462934 extrn __imp_GetOpenFileNameA:dword ; Create an open common dialog box
.idata:00462938 extrn __imp_GetSaveFileNameA:dword ; DATA XREF: GetSaveFileNameA
.idata:00462938 ; Create a Save common dialog box
.idata:0046293C extrn __imp_ChooseColorA:dword ; DATA XREF: ChooseColorA
.idata:0046293C ; Create a Color common dialog box
.idata:00462940
.idata:00462940
.idata:00462940
.idata:00462940 end start
```

Here's how the combined use of IDA and WinGraph32 allows the intuitive representation of a large amount of information from complex executables.

This tutorial is © DataRescue SA/NV 2005

Revision 1.1

[DataRescue SA/NV](#)

40 Bld Piercot

4000 Liège, Belgium

T: +32-4-3446510 F: +32-4-3446514