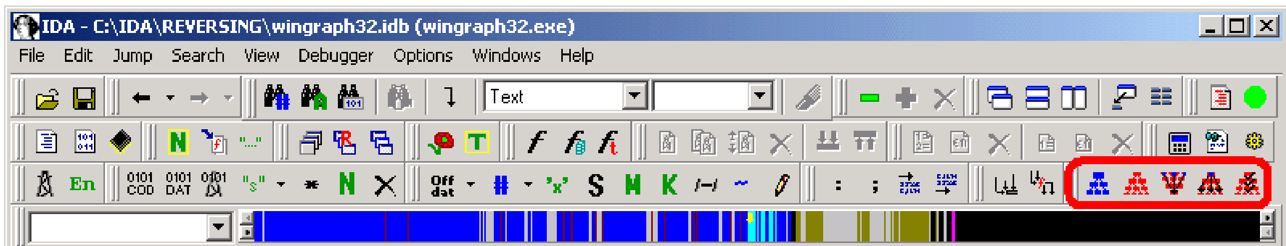
Graphing with IDA Pro.

© 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.

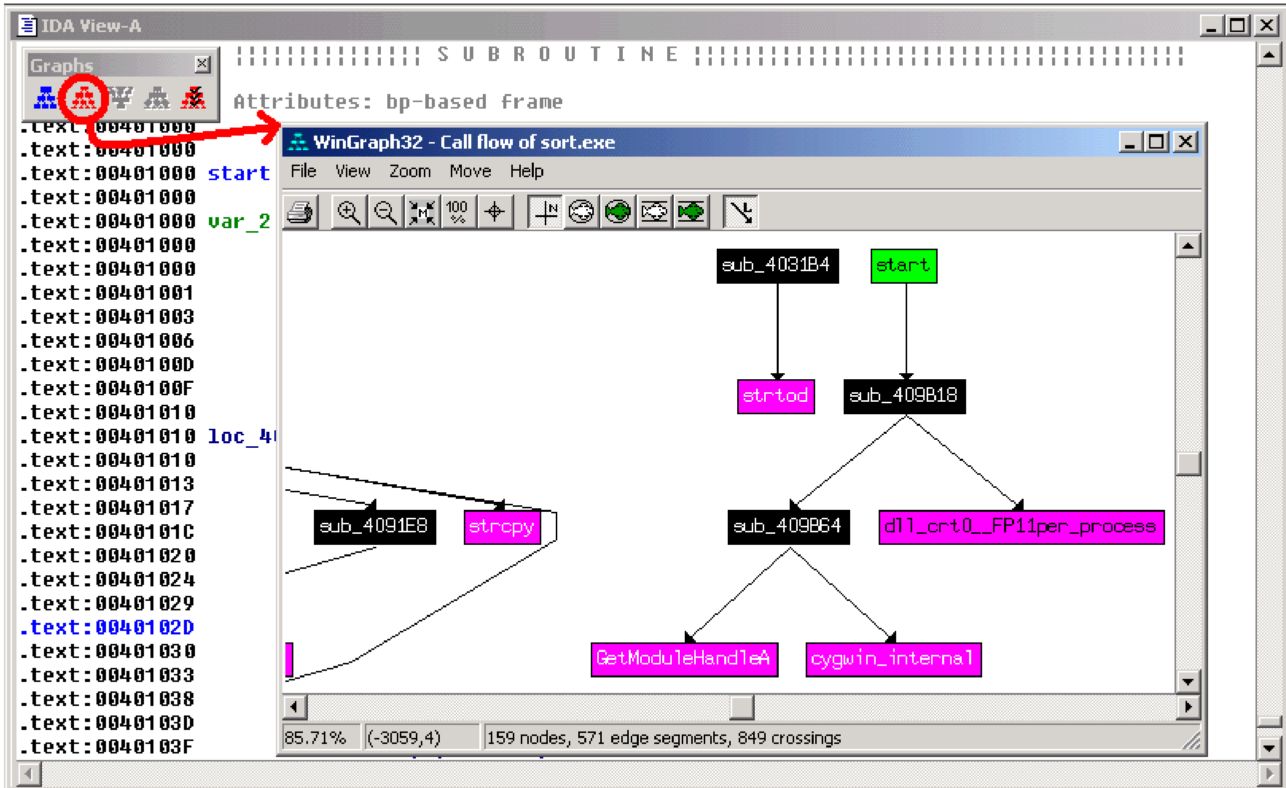
Flow charts.

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.

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.

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.

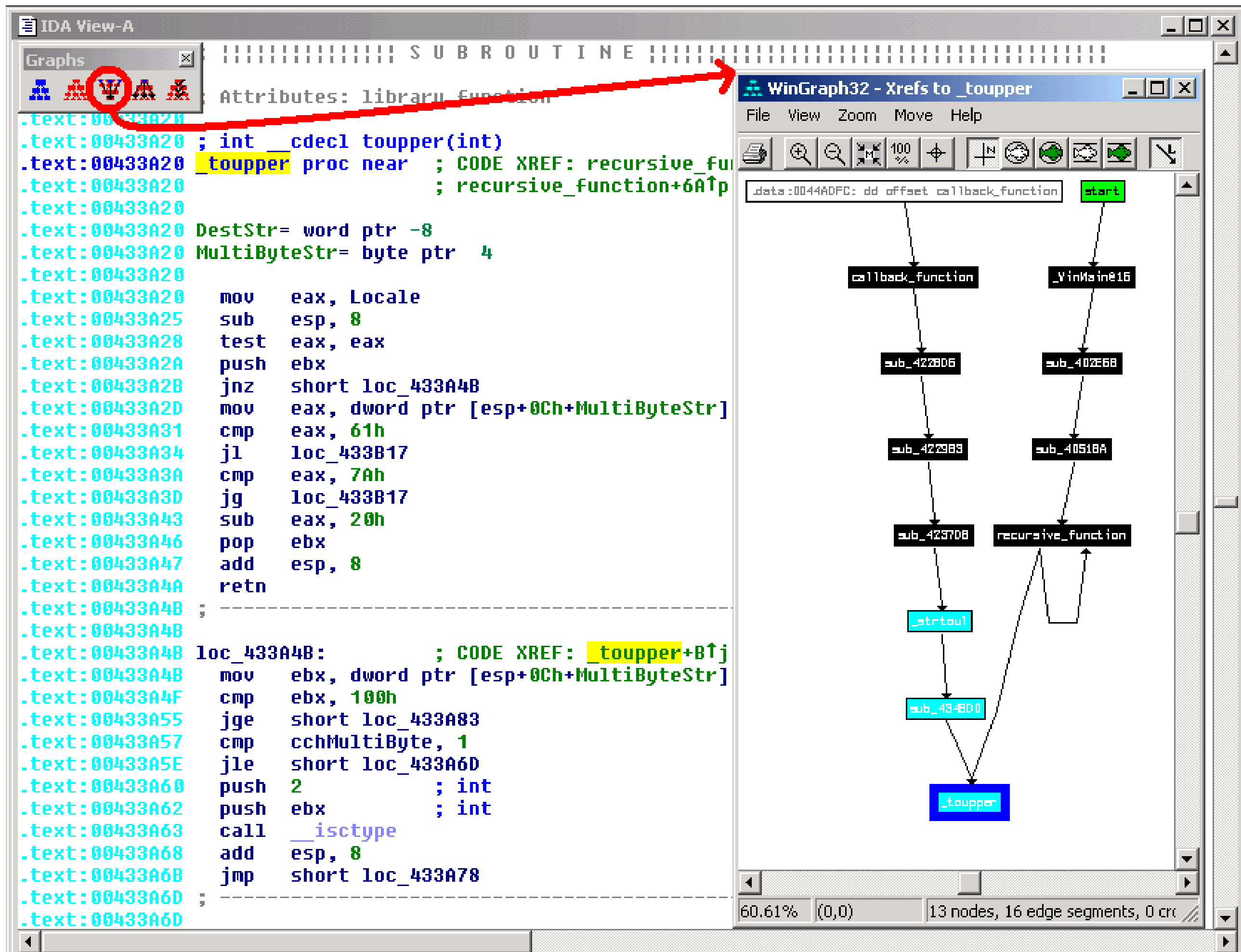


This command is mainly interesting for small programs, because the graph quickly becomes extremely complex in the case of a program containing a large number of functions. Notice that the graph's colours depend on functions/addresses attributes (externals functions, libraries functions, entry point,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



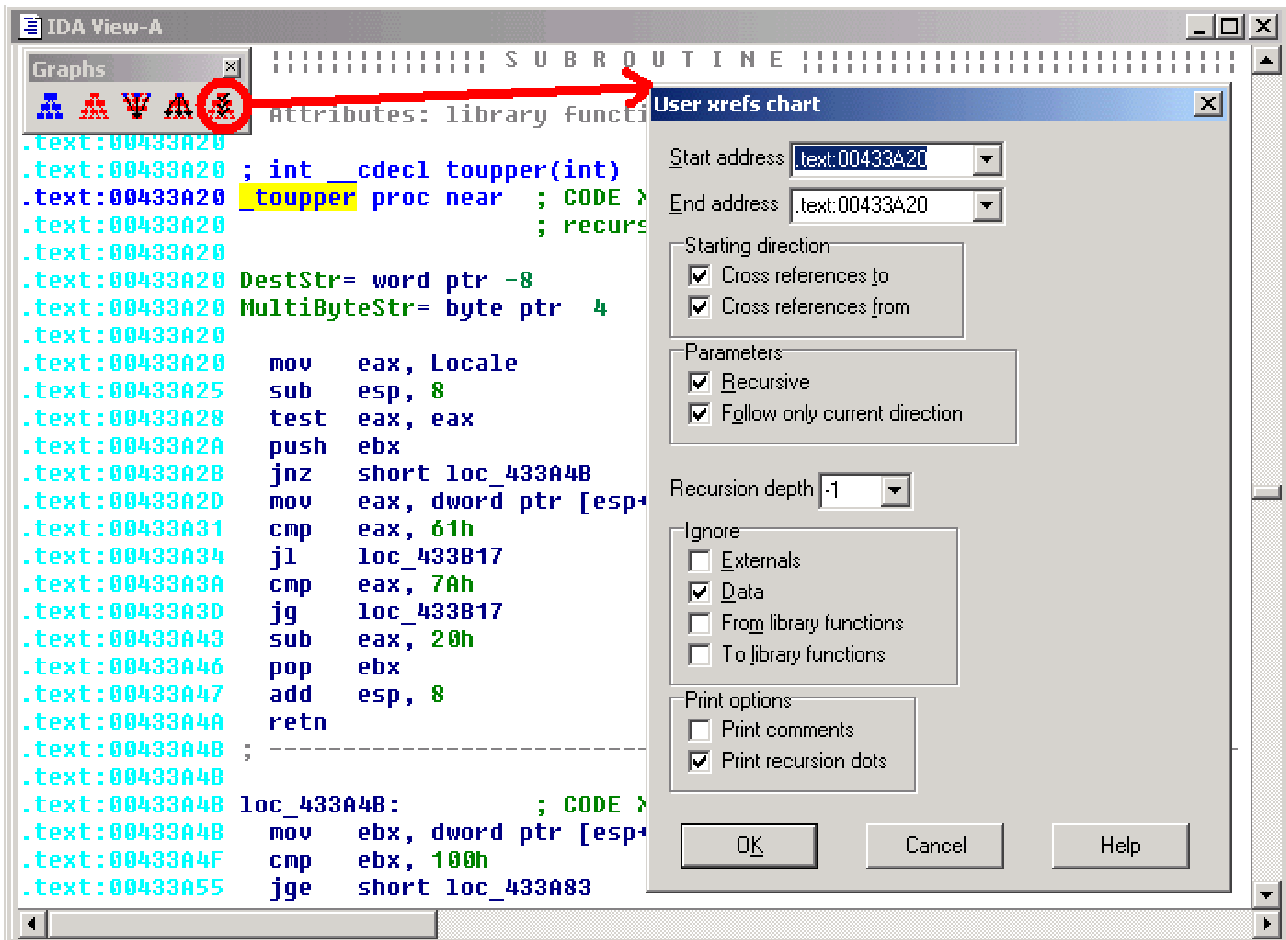
At the top of the graph, we notice the *start()* function: it calls the *WinMain()* function. On the left, we observe a function pointer to the *callback_function()*. On the bottom right, we can observe the *recursive_function()*, calling itself. We also remark that the *strtoul()* function (from the 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.

The *Xrefs from* command graphs code cross references coming from a given function. In this direction, only code cross references are used (and not data cross references) to avoid filling the graph with all the global data referenced from each function on the graph. Let's use this command on the *toupper()* function.

Here, we observe that the *toupper()* function make a wide use of others C Standard Library functions, themselves directly or indirectly using some Windows APIs, like *WideCharToMultiByte()* or *VirtualAlloc()*. As with the *Flow chart* command, those cross reference commands can also be used on a selection, to graph several functions.

User defined cross references chart.

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

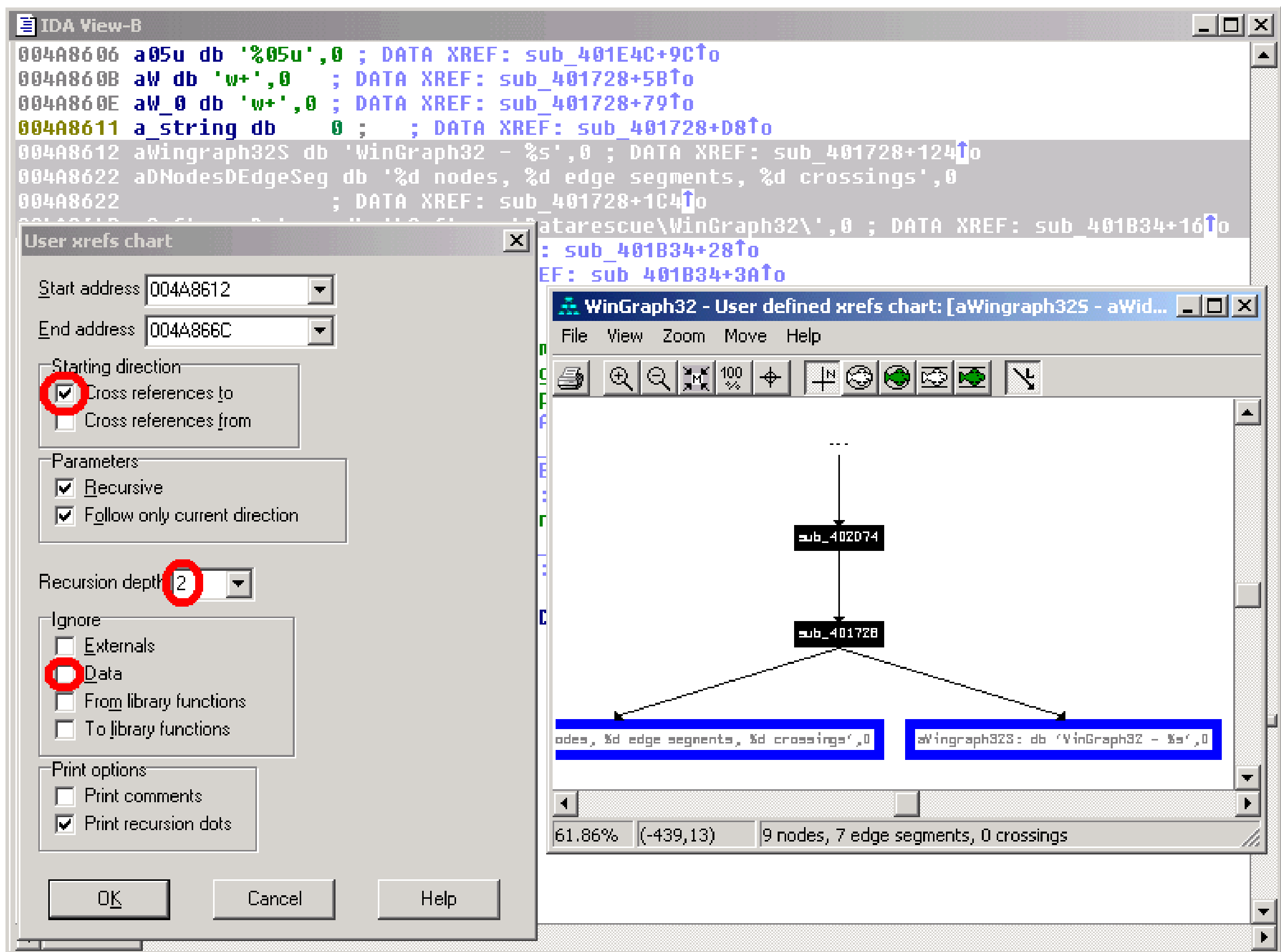


Let's take some practical cases requiring the use of these options.

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.

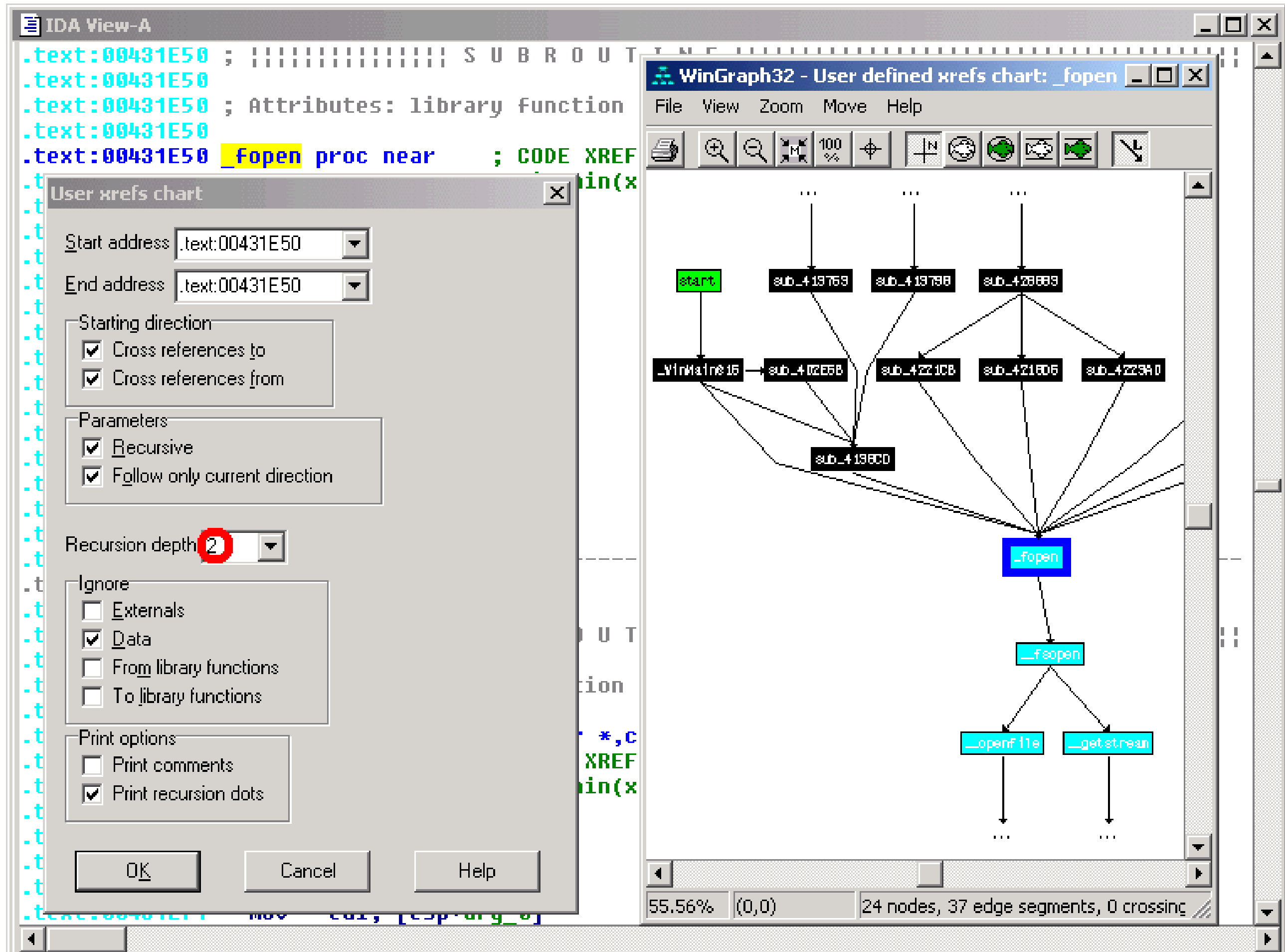
Yet another possibility is to show references to global data. We can show references to global data from a function, or search for functions referencing a set of global data referenced by the cross referenced functions. It is a good idea to specify the maximum recursion depth, to avoid an almost unreadable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

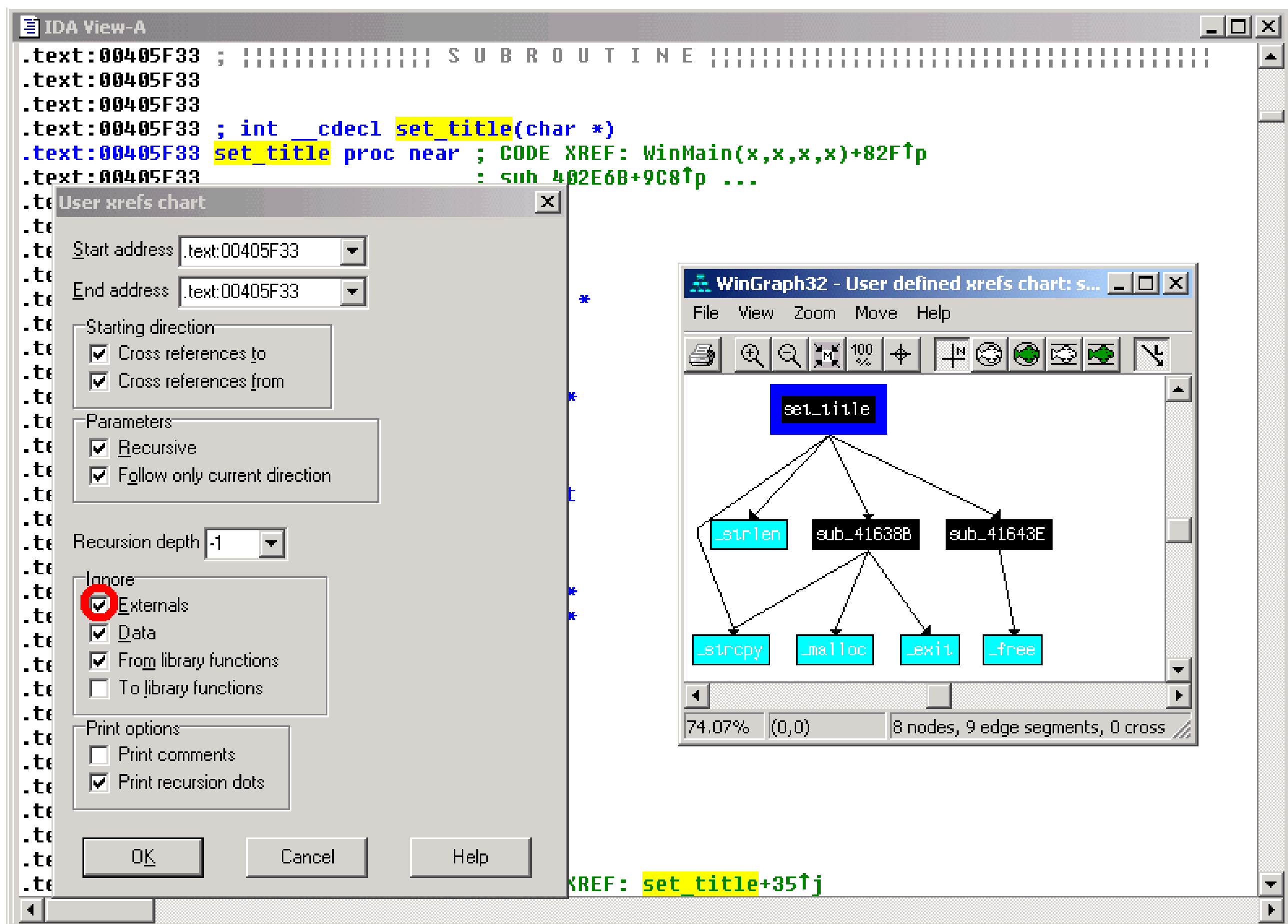
As we mentioned previously, the *Function calls* command isn't useful if used on a normal program containing a large number of functions. Again, by specifying a *Recursion depth*, we can try to get the same style of graph, but focused on a particular function. Let's try this on the *fopen()* function.



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.

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.

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*.

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