

How to use NCTUgr API

Wen-Hao Liu
dnoldnol@gmail.com
Latest updated: 2014/02/20

1. Introduction

In advanced technology nodes, routability has become a critical issue. To tackle this issue, many recent researches use fast global routers to do routability estimation. Accordingly, the routability of a design can be measured by the total wirelength, total overflow, and via count in its global routing result. To advance the routability-related research, global router **NCTUgr**'s API is released so that researchers can integrate NCTUgr into their tools to do routability estimation.

NCTUgr has two modes: regular mode [1] and fast mode [2]. The released API is the fast mode that is much faster than the regular mode but sacrifices little routing quality. Figure 1 shows the design flow of NCTUgr. Figure 1(a) shows the input of NCTUgr, that includes a 3D grid graph with the specified capacity to each grid edge and the pin locations of a set of nets. After the input is fed into NCTUgr, NCTUgr first packs the 3D grid graph into a 2D graph (Fig. 1(b)) and then identifies a 2D routing result (Fig. 1(c)). Finally, a 3D routing result is obtained by a layer assignment algorithm (Fig. 1(d)). In Figs. 1(a) and 1(b), the numbers next to grid edges denote the capacities of the grid edges. Figure 1 only gives a rough overview of NCTUgr, more details of NCTUgr are available in [1,2].

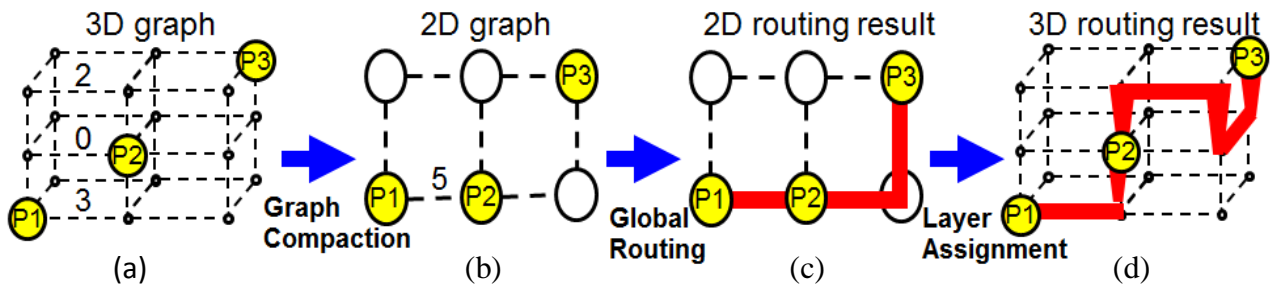


Fig. 1. (a) Input of NCTUgr; (b) a 2D grid graph; (c) a 2D routing result; (d) a 3D routing result

2. User Guideline

The downloaded package of NCTUgr's API includes **README**, **Router.a**, **main.cpp**, **Makefile**, and a set of *.h files in folder **[header]**. Router.a is the static library of NCTUgr, main.cpp is a simple program that invokes NCTUgr's API. If Router.a, main.cpp, Makefile, and [header] are put in the same folder, users can use command **make** to compile main.cpp with Router.a to get an execution binary. Users can read main.cpp to learn how to invoke NCTUgr's API in their source code, and then read Makefile to learn how to compile their source code with Router.a.

The code in main.cpp is shown as below, and the functionality of each NCTUgr's API is explained following the code.

```

=====
#include "header/interface.h"
void initInputForRouter(NetDB &netdb);
void userDefinedParameter(ParamSet &par);
int main( int argc, char *argv[] )
{
    //Initialization
    NetDB netdb ;
    initInputForRouter(netdb);           //Users can modify this function to set input for NCTUgr
    ParamSet par;
    setDefaultParameter(par, false);
    userDefinedParameter(par);          // Users can modify this function to control the routing flow of NCTUgr
    settingAndChecking(netdb, par, true);

    //Perform routing
    GlobalRouter gr(netdb.GridX, netdb.GridSizeY, netdb.layer, netdb.horCap, netdb.verCap);
    main_congestion_estimator( netdb, par, gr, NULL ) ;

    //Extract congestion information from the routing result
    printf("WL=%d TOF=%d Via=%d time=%.2lf\n", gr.totalWL, gr.tof, gr.totalVia, gr.totalRuntime);
    // Call functions getGapOfDemAndCap, getOverflowOfNet, getPassingNet, getWLOfNet

    //Clear the routing result
    clearRoutingResult(netdb);
}
=====

```

```
#include "header/interface.h"
```

When NCTUgr's API is required to use, please include file "interface.h".

```
NetDB netdb ;
```

NetDB is a data structure to store the input of NCTUgr. Namely, it stores the dimensions of a 3D grid graph, the capacity of each grid edge in the graph, and the pins' locations of a set of nets.

```
initInputForRouter(NetDB& netdb);
```

This function initializes the input for NCTUgr. Please refer to Section 2.1.1 to get more details.

```
ParamSet par;
```

ParamSet is a data structure to store a set of tunable parameters for NCTUgr.

```
setDefaultParameter(ParamSet& par, bool useFlute);
```

This function sets the default values for every parameter in **par**. Note that please always set **useFlute** to be *false*.

```
userDefinedParameter(ParamSet& par);
```

This function adjusts the values of some parameters in **par** to control the routing flow of NCTUgr.

Namely, users can control the routing flow of NCTUgr by modifying the content of this function. Please refer to Section 2.1.2 to get more details.

```
settingAndChecking (NetDB& netdb, ParamSet& par, bool showWarning);
```

This function initializes some built-in parameters in **netdb**, and then check whether the setting of **netdb** and **par** is correct. If there is something wrong, the errors are reported and then program terminates. If **showWarning** is set to *true*, the warnings are reported. These warnings usually do not hurt the routing process of NCTUgr, but it is better if no warning exists.

```
GlobalRouter gr(int gridX, int gridSizeY, int layer, vector<int> horCap, vector<int> verCap);
```

NCTUgr is initialized to create a 3D grid graph for routing. The dimension of the 3D grid graph is **gridX×gridY×layer** and the default capacities of each horizontal and vertical grid edge are respectively stored in **horCap** and **verCap**. Please refer to Section 2.1.1 to get more details about **horCap** and **verCap**.

```
main_congestion_estimator(NetDB& netdb, ParamSet& par, GlobalRouter &gr, char* outFile);
```

This function performs a light global routing engine [2] to quickly get a global routing result. Note that please set **outFile** always to be *NULL*.

```
printf("WL=%d TOF=%d Via=%d time=%.2lf\n", gr.totalWL, gr.tof, gr.totalVia, gr.totalRuntime);
```

After NCTUgr obtains a routing result, the values of total wirelength, total overflow, and total via of the routing result are respectively stored in variables **gr.totalWL**, **gr.tof**, and **gr.totalVia**, and the runtime of NCTUgr is stored in **gr.totalRuntime**. User can measure the routability of a design based on the values of these variables.

```
// Call functions getGapOfDemAndCap, getOverflowOfNet, getPassingNet, and getWLOfNet
```

Users can extract the congestion information from the routing result of NCTUgr by these functions. Please refer to section 2.2 to get more details.

```
clearRoutingResult(NetDB& netdb);
```

This function clears the routing solution of each net.

2.1 Input Parameter Setting

Users can modify data structure **NetDB** to initialize the input for NCTUgr, and modify data structure **ParamSet** to control the routing flow of NCTUgr. Sections 2.1.1 and 2.1.2 introduce the important variables in **NetDB** and **ParamSet**, respectively.

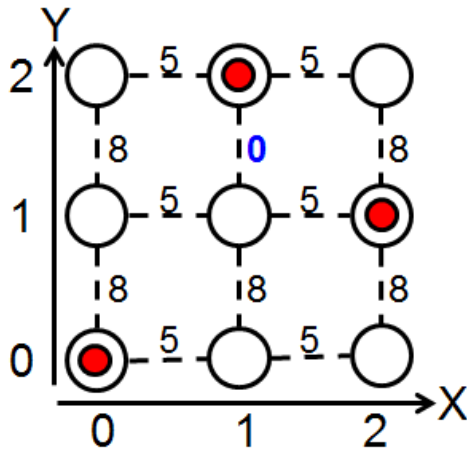
2.1.1 Input Setting

Data structure **NetDB** has following variables to represent the input for NCTUgr.

```
struct NetDB {
    Int gridX, gridY, and layer;
    vector<int> horCap, verCap;
    vector< AdjustCapEdge > adjArray;
    vector<Net> netArray ;
}
```

- The variables **gridX**, **gridY**, and **layer** denote the x-, y-, and z- dimensions of a 3D grid graph, respectively.
- The vectors **horCap** and **verCap** respectively denote the default capacity for the horizontal and vertical grid edges in each layer of the 3D grid graph. For example, horCap[0]=10 mean that the default capacity of the horizontal grid edges in the first layer is 10.
- The vector **adjArray** represents a set of grid edges whose capacities are not default due to the blockages or local congestion. Each element in adjArray has 5-tuple $(x, y, z, hori, value)$, where *hori* is a boolean value and others are integers. An element $(x_1, y_1, z_1, true, v_1)$ in adjArray denotes that the capacity of the grid edge between nodes (x_1, y_1, z_1) and (x_1+1, y_1, z_1) is v_1 , while another element $(x_2, y_2, z_2, false, v_2)$ in adjArray denotes that the capacity of the grid edge between nodes (x_2, y_2, z_2) and (x_2, y_2+1, z_2) is v_2 . Note that the specialized capacity cannot be greater than the default capacity. For example, if an element $(x_1, y_1, z_1, true, v_1)$ is in adjArray, v_1 cannot be greater than horCap[z_1].
- The vector **netArray** represents a set of nets, in which each net has a **id** and a vector **pin3D** to store its pin locations.

An example is shown in Fig. 2 to illustrate how to represent an input for NCTUgr, in which the dimension of the 3D grid graph is 3×3×1, the number next to each grid edge denotes the capacity of the grid edge, and the grid graph contains a three-pin net.



```
NetDB netdb;
netdb.gridX=3, netdb.gridY=3, netdb.layer=1;
netdb.horCap.push_back(5);
netdb.verCap.push_back(8);
AdjustCapEdge e;
e.x=1, e.y=1, e.z=0, e.hori=false, e.value=0
netdb.adjArray.push_back(e);
Net n;
n.id=0;
n.pin3D.push_back(Point(0, 0, 0));
n.pin3D.push_back(Point(1, 2, 0));
n.pin3D.push_back(Point(2, 1, 0));
netdb.netArray.push_back(n);
```

Fig. 2. An example to illustrate how to represent an input for NCTUgr by NetDB

2.1.2 Parameter Setting

Data structure **ParamSet** has following variables to control the routing flow of NCTUgr.

```
struct ParamSet {  
    Int PtRounds, MnRounds, RarRounds.  
    Int laType;  
    bool print_to_screen;  
}
```

- NCTUgr consists of three routing stages: L-shape pattern routing, monotonic routing, and ripup and rerouting. The variables **PtRounds**, **MnRounds**, and **RarRounds** in ParamSet control the routing iterations of these stages. More routing iterations can yield a routing result with lower total overflow but spend longer runtime. More details about each routing stage are available in papers [1, 2].
- The variable **laType** decides which algorithm is used in the layer assignment stage. If laType=0, NCTUgr does not perform layer assignment; if laType=1, a greedy algorithm is used; if laType=2, a dynamic-programing-based algorithm is used. Note that, to do fast routability estimation, laType is suggested to set 0 for time saving but via count information will not be obtained. If users would like to get the routability estimation result including via count information, laType is suggested to set 1 or 2. (The dynamic-programing-based algorithm gets fewer via count but costs longer runtime than the greedy algorithm).
- If the variable **print_to_screen** is *true*, the routing log of NCTUgr will show on the screen; otherwise, the log is invisible.

2.2 Extracting Routing Congestion Information

NCTUgr's API provides several functions to extract the routing congestion information from the 2D routing result obtained by NCTUgr (Fig. 1(c)) so that users can optimize the routability of their designs based on the extracted information. The reason of extracting the routing congestion information from the 2D result (Fig. 1(c)) rather than the 3D result (Fig. 1(d)) is because the 2D routing result can be regarded as a condensed 3D routing result, so the routing congestion information in the 2D routing result provide a global view across multiple layers.

```
int getGapOfDemAndCap (GlobalRouter& gr, int x, int y, bool hori);
```

If **hori** is *true/false*, this function returns the value of $d(e)-c(e)$ of the grid edge e between the nodes (x, y) and $(x+1, y)/(x, y+1)$ in the 2D grid graph, where $d(e)$ denotes the number of routing nets passing through e , and $c(e)$ denotes and the capacity of e .

The time complexity of this function is $O(1)$.

```
int getOverflowOfNet(GlobalRouter& gr, Net& n);
```

If $d(e)$ is greater than $c(e)$, the grid edge e is defined to be overflowed edge. This function returns the number of overflowed edges passed by net n .

The time complexity of this function is $O(L)$, where L denotes the routing wirelength of net n .

```
vector<int> getPassingNet(GlobalRouter& gr, vector<Net>& netArray, int x, int y, bool hori);
```

If **hori** is *true/false*, this function returns the ids of a set of nets passing through the grid edge between nodes (x, y) and $(x+1, y)/(x, y+1)$. For example, if three nets whose ids respectively are 0, 10, and 15 pass through the grid edge e between nodes $(5, 2)$ and $(6, 2)$, `getPassingNet(netArray, gr, 5, 2, true)` will return a vector $\{0, 10, 15\}$.

The time complexity of this function is $O(N_e)$, where N_e denotes the number of nets passing through grid edge e .

```
int getWLoFNet(Net& n)
```

This function returns the wirelength of net n .

The time complexity of this function is $O(1)$.

3. Discussion

When users use NCTUgr's API, the dimension of the grid graph and the capacities of grid edges have to be specified by themselves. However, how to properly set these variables is an open and critical problem. When the dimension of the grid graph is set small, the local routability can be more accurately estimated but NCTUgr will spend longer runtime. Also, if the capacities of grid edges are well modeled to consider the effect of vias, local blocks, and pin access [3], the routability estimation by NCTUgr can be more accurate.

4. Reference

- [1] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, Kai-Yuan Chao: NCTU-GR 2.0: Multithreaded Collision-Aware Global Routing With Bounded-Length Maze Routing. *IEEE Trans. on CAD of Integrated Circuits and Systems* 32(5): 709-722 (2013).
- [2] Wen-Hao Liu, Yih-Lang Li, Cheng-Kok Koh: A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing. *ICCAD 2012*: 713-719.
- [3] Wen-Hao Liu, Cheng-Kok Koh, Yih-Lang Li: Case study for placement solutions in ispd11 and dac12 routability-driven placement contests. *ISPD 2013*: 114-119.