ESE556: VLSI Design Automation and CAD

Project 2: Simulated Annealing Placement Algorithm Implementation

Project completed by Lars Folkerts

11/9/2015

Introduction

This project discussing several data structures to be used in a chip placement simulated annealing algorithm. The desire of this placement algorithm is to minimize interconnect distance between chips and prevent overlap in a time efficient manner. The implementation of these data structures and simulated annealing algorithm still faces some issues at the time of writing this, and more work will need to be done to overcome these issues.

Problem Statement

The problem we face in placement is to generate an efficient algorithm that minimizes interconnect distance, overlap and out-of-bounds (OOB) modules. The simulated annealing algorithm is a good place to start with this algorithm due to its success in the past.

The simulated annealing algorithm moves a node on the grid and determines the cost of that move. Here, cost is function representing the 3 minimized parameters, each with their own weights. If the cost results in an overall cost reduction, the move is accepted. Otherwise, it is accepted with a probability $e^{-d(\mathbf{cost})/T}$, and else rejected. Here, d(cost) is the change in cost and T is temperature. This is performed for several hundred iterations before temperature T is reduced by alpha. The idea is that in the first several iterations, bold moves are allowed as the nodes try and figure out what the best solution is; as time goes on, only more stright forward moves are allowed as nodes are trying to find their placement.

Previous Research

One of the best simulated annealing algorithms is the TimberWolf algorithm. This algorithm uses the cost function:

$\Sigma(x_{interconnect}) + \Sigma(y_{interconnect}) + \Sigma(overlap)*OVERLAP\_WEIGHT + \Sigma(Out\text{-}Of\text{-}Bounds)*OOB\_WEIGHT$

and accordinding to TimberWolf 3.2, the following parameterization:
I**NIT_TEMPERATURE** 4000000
FINAL_TEMPERATURE 0.1
0.8 < **ALPHA** < 0.95
Modules/5 < **NUM_ITERATIONS** < Modules/2

where alpha is greatest in the mid temperature range. The number of iterations depends on the size of the circuit and a lookup table is perhaps best to use for this implementation.

(Sherwani 226-228)

Data Structures

The high level view of the data structures used in this project is presented in this section. However, it is first important to look at the file contents so that the reader can browse through the code as they read the following sections

File Structure

algo.c - This file contains the main function and the simulated annealing algorithm
generate_data - This module helps the main function generate the data structures. It htakes file descriptors as inputs.
grid - This contains the data structure for our non-overlapping grid implementation
overlapgrid - This contains the data structure for our overlapping grid implementation
node - This module contains all node information, graph manipulation and cost functions
helper - This module contains miscellaneous functions, mostly to help generate_data read inputs
image - This module uses the ImageMagick library to create a picture output

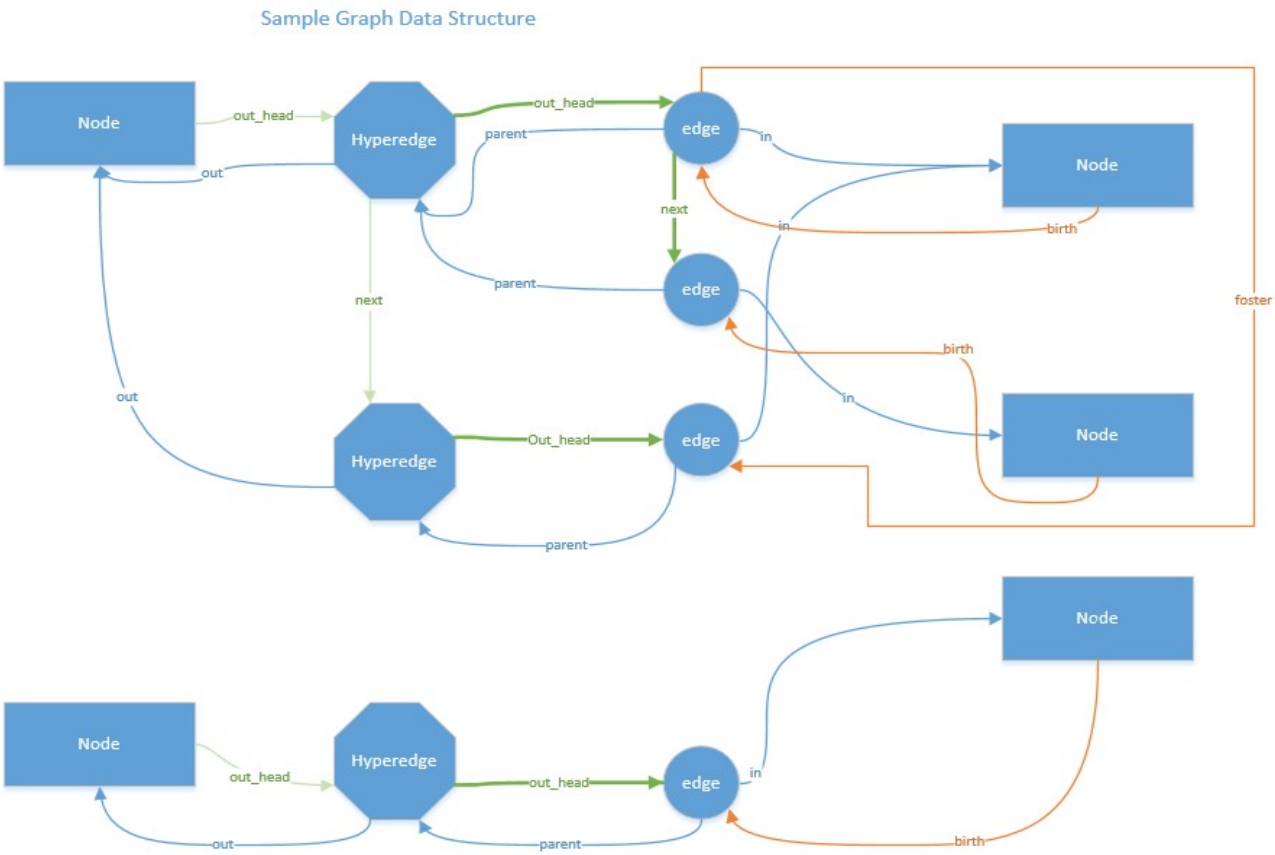
parameters.h - This contains all the preprocessor parameters used in our simulated annealing algorithm, and can be used to tweak our algorithm once it is fully implemented

Graph

To represent the graph, 3 data structutres were used: node, hyperedge and edge.
Nodes were the pads and cells in the assignment and what is being partitioned;
Hyperedges are node outputs. They contain from 1 to many edges that will link to another node.
Edges are node inputs. They connect the hyperedge node outputs to the following node. They alsoti contain a linked list structure of inputs into their output node (named node.birth and edge.foster).

The diagram on the next page, created with MS Visio, perhaps best illustrates the relationship between these components.

Only one copy of the graph is generated. The N_ArrCpy nodes use graph pointers that point to N_Arr nodes. This is only OK because the graph is not changing in between placement iterations. Thus we do not need to backup this information and we can simply us N_Arr's graph information for calculating cost.



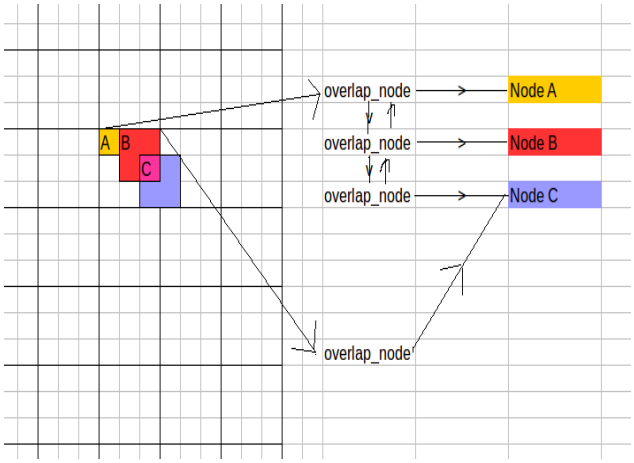

```
struct node
{
        char type; //padi/terminal (p) or cell (a)  or empty (e)
        int index; //pad or cell #
        int locked; //counter to LOCK_THRESH
        int cost;
        struct edge* birth; //first edge input
        struct hyperedge* out_head; //hyperedge output
        char dir; //direction (pads only- other nodes have more than one pin)
        char orientation;

        int x; //x coordinate
        int y; //y coordinate
        int width;
        int height;
        //corner stitching
        struct node* north;
        struct node* south;
        struct node* east;
        struct node* west;

};
struct edge
{
        struct node* in; //the node this hyperedge connects to as an input
        struct hyperedge* parent; //the parent edge of this hyperedge
        struct edge* next; //next out node this net connects to
        struct edge* foster; //next input hyperedge this out node is connected to
};
struct hyperedge
{
        struct node* out; //the node this edge connects to (output signal)
        struct edge* out_head; //head of a linked list of hyperedges this edge expands to
        struct hyperedge* next; //next output the in node is connected to
};
```

Overlap Grid

For a grid that allows overlapping, an easy data structure is a bin-based grid. The OverlapGrid data structure takes the layout grid and divides it into midsize bins. The bin height is equal to the row height, and the bin width is a compile time parameter GRID_GRAIN=64. Each bin can contain several nodes, and these nodes can overlap each other. Thus the grid s actually a series of pointer to a node-wrapper structure, overlap_node. This structure is a double linked list of nodes.



The basic form of this data structure is shown to the in Figure Overlap 1. We have inserted 3 nodes into 2 bins. The head of the linked list in the left most of these 2 bins list is arbitrarily an overlap_node that points to Node A. In reality, the linked list is a stack, so the most recent node inserted is the head. B and C are also located in this bin, so they are also in this linked list.

The next bin only contains Node C, so we have only a singe overlap_node that points to Node C.

Insertion

Insertion into this grid relatively easy, since all we need to do is append the node into the overlap_node linked list. The real difficulty comes in calculating the change in overlap for the insertion. The inserted node will look in each bin for other nodes. If it finds one it has not yet encounter, in calculates the amount of x-overlap and amount of y-overlap, then multiplies these two 1-dimensional values together to get the 2-dimensional overlap. This value squared is then added for to the total overlap for the insert function.

A simple doubly linked list is used to keep track which nodes the inserted node has already encountered. This is important so that we do not calculate overlap more than once. If we are certain that we will not encounter a node again, i.e. we have already visited all of the bins that it has in common with the inserted node, we can remove it from the list.

The insertion also calculates area of a node that is out of bounds. Two out-of-bounds (OOB) rows and two OOB columns are added to the end of our bin list for this case. The first OOB row:column of bins are for nodes that extend passed the south:east boundary of the grid. The second OOB row:column of bins is for the nodes that are too far north:west of the boundaries, i.e. y<0:x<0.

Removal

Removal is almost the same code as insertion with calculating x and y overlaps. A common function named work_overlap was used for both functions. This function took in a parameter called insert_flag, which will remove a node when it is equal to 0 and insert a node otherwise.

MoveOverlap

Since we do not care about overlap when moving a node, only on move function was created for the overlap grid. This function will remove a node, generate a random placement, insert the removed node in this placement and return the change in cost according to the Timberwolf algorithm.

Accept/Rejecting a Move

A backup copy with the suffix Cpy is maintained for all OverlapGrid data structures. This is the N_Arr of nodes and the OverlapGrid of overlap_node*. If we accept a move, AcceptOverlapMove() is called which copies the working copy to the backup copy. If we accept a move, RejectOverlapMove() is called which copies the the backup copy the working copy, resotring its original state.

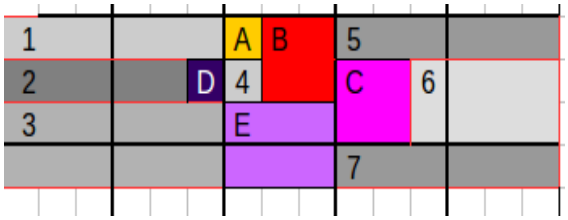
The nodes can be copied directly since they contain only values i.e. no pointers to memory locations. Thus we do not need to worry about the address of node copared to nodecpy. However, for the overlap_node structures in the grid, we do need to worry about this. We use the type and index of a node to locate it in N_Arr or N_ArrCpy in order to work across these two copies mutually. The overlap_node structure has no ID and is an arbitrary wrapper, so we delete, add and manipulate these structures when moving through the linked list without concern for their memory locations.

The logic in these functions is currently non-functional, as proven with the verify() debug function. We find that

overlap_nodes are not successfully removed from a grid. After a few iterations (~20 accept/rejects), a node cannot find itself in order to remove itself from the grid, and an assert statement in remove_node()->work_overlap() fails. This was as far as a I got with my code.

Grid (Non-Overlapping)

A second grid variation was implemented but is untested. This grid enforces strict rules, such as staying within the boundaries and having no overlaps. We used the corner stitching data structure for this grid, but we also have a grid of node pointers overlayed on top of these nodes for quick access of nodes when inserting/removing these nodes into the grid.



The figure shown right shows part of a the grid. Empty nodes are shown in grey and are numbered for ease of discussion. Each node has 4 pointers. From the north-east most point, there is a north pointer and an east pointer. For example, for Node E, the north pointer goes to Node B and the east pointer goes to Node C. On the south-west corner, there are the other two node pointers, south and west. For example, Node B's south pointer goes to Node E and its west pointer points to Node 4. By using this structure, we can navigate through the grid locally and find out what is in the vicinity of our node.

For large scale movement, the superimposed grid is in place to get us in the vicinity of our target node. This grid is a 2-dimensional array of node pointer. These node pointers point to the node in the NE corner of its bin. This is done in the find() function. For example, the center Grid square in the figure points to A; the next index to the right points to empty Node 5; the next index below node A points to Node E.

It is also important to note that this data structure creates empty nodes horizontally only. This means that the longest horizontal is chosen for an empty node; nodes are “merged” verically only if possible after the this constraint. For example, we cannot combine empty nodes 1 and 2 above, and have 1x1 node to the left of A; for this case, neither node 1nor the 1x1 node are as wide as they can be. This consistency simplifies the code and analysis dramatically.

The empty nodes are stored in a large array for ease of access and copying. They are compacted to the start of the array. A hash table would be better for access time and copying, but time was not there to implement a good hashing function.

Insertion

The InsertNode() function first checks to see if an area is clear to insert a node in. If the area is occupied by another non-empty node, the function does not insert the node and returns the node that is blocking. Upon a successful insertion, NULL is returned and the int* err argument is equal to 0. err is nonzero for an out-of-bounds (OOB) exception.

To maintain the grid structure, an expansion node is created that expands downward as empty nodes are shrunken or removed to make way for the inserted node. The progression downward across the insert area occurs iteratively, as we shrink or remove one empty node at a time to make way for the inserted node. The expansion node holds the void space to maintain grid navigat(e)-ibiliy.

The empty nodes we are inserting over are guaranteed to have the same horizontal top line. This means there are only four cases to consider for insert:

1. Empty node is expands past the area width on both sides
   For this case, we shrink the empty node westward and creat a new empty node to the east
2. The inserted node has a module node to its east, but the west is an empty node-wrapper
   For this case, we simply shrink the empty node westward
3. The inserted node has a module to its west, but not to its eastward
   For this case, we shrink the empty node eastward
4. The inserted node is neighbored by nodes to its east and west border
   For this case, we remove the empty node

As we will see next, this is much simpler than removal, where the east and west horizontal boundaries are not guaranteed to be on the same longitude and latitude.

One final note is that the GridLock integer is used to mark critical sections of code where the grid structure is temporarily broken. It is mostly for the coder to see that he should not rely on the grid pointers (and that he must restore the grid pointers) in between the GridLock++ and GridLock-- sections. The find() function asserts that GridLock==0 so that we can identify the problem quickly.

Removal
Removal of a node differs from insertion for this data structure.
First, we are guaranteed to remove a node, so we do not need any checking to take place as we did in insert.
Next, a similarity. Similar to inserts' expansion node, a filler node takes the place of the node to be removed. This maintains the grid structure throughout the removal. We also use the GridLock variable to mark critical sections of code where the grid structure is broken.
Thirdly, we must move along the east and west boundries of a node simultaneously for node removal. Using the sample grid on the previous page, in order to remove E we would first need expand and partially merge nodes 3 and 7; then we would need to consider nodes 3 and C. The problem with this is that we can only move clockwise around the boundry of a node. This means that we can only move north along the west border and south along the east border. To account for this, we use the combine_ew() recursive function. It fills up the call stack moving south along the removed node's east border. It then returns northward. Meanwhile, the current west node we are working with is returned from combine_ew(). Thus when the line of latitude we are working at moves above west, we return west->north to move up the west boundary.
Fourthly and perhaps the most difficult to deal with was all of the potential cases of removals. There were 12 cases to consider. The four cases from insert still exist; that is

1. Empty nodes are to the east and the west of removal
2. There is a module eastward of the removed node, but the west boundary is an empty node
3. There is a module westward of the removed node, but the east boundary is an empty node
4. There are modules both eastward and westward of the removed node

Each of these 4 cases have 3 subcases, which leads the total of 3*4=12 cases

1. The east and west nodes have the same northern boundary
2. The east boundary is higher than the west boundary
3. The west boundary is higher than the east (e.g. nodes 3 and 7 when removing E)

It would be too long to discuss all 12 subcases here. The code comments explain the appropriate actions.

MoveRandom
This function picks a random location to insert a node. If the node is not able to fit in this new random location, MoveLocal() is called to shift the blocking node until the area is clear.
This function was meant for a full non-overlapping simulated annealing algorithm at high temperatures. It allows for dramatic changes.
MoveLocal
This function accepts a node to move and a keepout area. The node that needs to moved randomly chooses north, south, east or west to move to the keepout area boundry. If this move would cause another collision and the moved node cannot be inserted in its new location, MoveLocal() is called recursively on the blocking node.
This function is default for moving nodes in the Grid datastrucure due to its simplicity and the preventing the change of many nodes for low temperature of the simulated annealing algorithm.
MoveShift
This function shifts a node slightly from the nodes current position. That is, it either move up one row, down one row, the the left one index, or to the right one index. If a node gets in the way of this shift, MoveLocal is called.
This function was introduced for the assumption that pretty good placement was already achieved. We can call this function after the running through the simulated annealing algorithm for overlapping, and trying to rinsert those nodes into the Non-Overlapping grid.
MoveEmpty
This function was an idea I had but was not implemented successfully. The idea was to create a data structure that sorts empty nodes into bins based on their width. Then, when a node needs to be moved with a guarantee of being accepted, we can select a random empty node from this list that is wide enough to accept the inserted node. We can then see if the height necessary to insert the node is also available. This method was too difficult to implement since the bins were hard to monitor when copying to/from the backup copy in accept and remove.

Accept/Rejecting a Move
These functions follow the same logic as the OverlapGrid equivelants. For this one, in addtion to copying N_Arr and the Grid, we also need to copy the empty nodes.
Copying N_Arr and the empty nodes was a little more difficult since the nodes in N_Arr have different memory locations than the nodes in N_ArrCpy, despite all of the non-pointer memory values being the same. Thus we created two new functions - CopyParallelNode and RestoreParallelNode() - for AcceptMove() and RejectMove() respectively. These functions represent
the “parallel universe” in which the node copies are stored. When copying the pointers over from one node to the

other, the node type and index are used to find the equivelant parallel node, which is stored in an array. N_Arr(Cpy) and EmptyNodeList(Cpy) are both arrays which leads to easy indexing of these nodes.
The Grid employs a parallel copying scheme for its pointers, but unlike OverlapGrid it is just a 2D array of pointer instead of a 2D array of bins. This leads to a much easier copying implementation.

Other Implementation Details

Cost

There cost function right now uses the Timberwolf algorithm, which is:
$\Sigma(x_{interconnect}) + \Sigma(y_{interconnect}) + \Sigma(overlap)*OVERLAP\_WEIGHT + \Sigma(Out\text{-}Of\text{-}Bounds)*OOB\_WEIGHT$
For the interconnect differences, we only calculate the difference from moved nodes. For overlap and overlap weight, we constantly monitor the cost when inserting and deleting a node in the OverlapGrid. For the Grid data structure (No Overlapping, No OOB), we do not need to monitor these functions since they are always 0.
The interconnect distance proposed some challenges for me. First, no points were assigned on a node for hyperedges to connect to. The code right now just uses the coordinate (x,y) as its interconnect. This will likely be changed to the center of the node once the code is running. Another option is to have the cost function use the best possible coordinate for each edge, but this is misleading since nodes sometimes belong to the same hyperedge.
The second thing to consider is net weights, information which was missing from the benchmark files we received. We therefore did not weight any of the edges, although this implementation would be simple if the information was provided to us.
The next thing to consider is the treatment of hyperedges. The current implementation assumes that each hyperedge can be treated as a series of edges, which inflates the interconnect distance. This is a simple, fast a good approximation. However, nets connecting in the same x direction can share the same net for part of their trip, for example. These are more routing details that would likely not concern us.

Results

The code is not working as of yet and time has run out. The OverlapGrid is capable of inserting and removing nodes and making several moves. However, it is confirmed with the verify() debug function at the bottom of the overlapgrid.c file that the AcceptOverlapMove() and RejectOverlapMove() functions are not copying the information over correctly.
The regular Grid structure is untested and will require some debugging as well.

The writing of an image is coded in image.c using the ImageMagick library. However, this was also not tested or integrated into the main() function. If there are problems compiling the code due to the linking and installation of ImageMagick, support for image.c can be removed.

The overall algorithm will require some testing and tweaking once the rest of the code is working. I am unsure of the cost function working correctly or even returning the right sign. More work and testing will need to be done in this area.

Conclusion

The scope of this assignment is huge, especially in the way I choose to implement it; perhaps it would have been better to limit the work I completed and to complete this assignment in smaller steps and in smaller scope. Mainly this would entail encoding one grid instead of two, or even simplifying the grid to a 2D-array of 1-width x 1-height node pointers, non overlapping. I was more concerned in this project about creating a fast and efficient algorithm that I forgot factor in the number of hours I was able to work on the project into the implementation details. Even given a significant amount of more time to work on the project, as it currently stands, there is a lot of issues to dive into in order to get a successful implementation.

The code base for this assignment is in the attached appendix. Any updates to the code or report can be found at https://github.com/lfolkerts/ESE556P2 . If I have time, I will continue to work on this code and if I get a copy I am happy with, I will re-email you the code.

Works Cited

Sherwani, Naveed. “Algorithms for VLSI Physical Design Automation”. 3rd Edition. Kluwer Academic Publishers. 1999.

Appendix: Code

Modules are listed alphabetically, with the exception od parameters.h which is shown here and Makefile, which is at the bottom.

parameters.h

```
#define TEST_DIR "test/"
#define SF_L 5+7+6+1 //test/ibmXX/ibmXX. and NULL terminator length
#define S2US 1000000 //seconds to microseconds
//#define SEED_RANDOM
#define INIT_X 0
#define INIT_Y 0
#define INIT_TEMPERATURE 300000
#define FINAL_TEMPERATURE 0.1
#define INIT_ALPHA 0.8
#define RAND_PRECISION 100000
#define GRID_GRAIN 64
#define OVERLAP_WEIGHT 5
#define ROW_WIDTH_WEIGHT 1

//#define OUTPUT_CSV
#define TIME_REPORT
#define DEBUG
#ifdef DEBUG
//VERBOSE DEBUGGING
// #define DEBUG_VB_ALGO //main module
// #define DEBUG_VB_GEN //generate module
// #define DEBUG_VB_OLG  //overlap grid module
#endif
```

algo.c

```
#include <unistd.h> //getopt
#include <getopt.h> //optind
#include <stdio.h>
#include  <fcntl.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <math.h>
#include <assert.h>
#include "parameters.h"
#include "generate_data.h"
#include "grid.h"
#include "helper.h"
#include "node.h"
#include "overlapgrid.h"

void sa_algorithm_no_overlap(int iterations);
void sa_algorithm_shift(int iterations);
void sa_algorithm_overlap(int iterations);
void update_iterations(int* iter);
void update_alpha(double *alpha);

/*********** main ***********
 * Overall wrapper function
 * processes argc and argv, opens/closes files
 *
 * returns 0 on success, negative errorcode otherwise
 ****************************/
int main(int argc, char **argv) {

        int return_code = argc;
        struct timeval tv_start, tv_init, tv_done;
        struct node* n;
        //command interpretation
        extern int optind;
        int arg_num; //maintaining control of command line options
        char opt; //command line options
        int i, f_no; //for loop counters
        int cost_org, cost;
        char sf_nodes[SF_L+5], sf_nets[SF_L+4], sf_pl[SF_L+2], sf_scl[SF_L+3], sf_wcl[SF_L+3];
        FILE *f_nodes, *f_nets, *f_pl, *f_scl, *f_wcl;

#ifdef SEED_RANDOM
        srand(time(NULL));
#endif
        gettimeofday(&tv_start, NULL);
        f_no = 1;

        /*Process through options */
        while ((opt = getopt (argc, argv, "th:")) != EOF)
        {
                switch (opt)
                {
                        case 't':
                                f_no = atoi(argv[optind]);
                                if(f_no > 18 || f_no <= 0){  fprintf(stderr, "Illegal Test Number %d\r\n",
f_no);  goto exit; }

                        case 'h':
                                Help();
                                return_code=0;
                                goto exit;
                                //break;
                        case '?':
#ifdef DEBUG
                                fprintf(stderr, "Error: unrecognized argument %s\r\n", argv[optind]);
```

```
#endif
                                RequestHelp();
                                return_code = -1;
                                goto exit;
                        }
                }
                /****************************** Open Files ********************************************/
                sprintf(sf_nodes, "%sibm%02d/ibm%02d.nodes", TEST_DIR, f_no, f_no);
                sprintf(sf_nets, "%sibm%02d/ibm%02d.nets", TEST_DIR, f_no, f_no);
                sprintf(sf_pl, "%sibm%02d/ibm%02d.pl", TEST_DIR, f_no, f_no);
                sprintf(sf_scl, "%sibm%02d/ibm%02d.scl", TEST_DIR, f_no, f_no);
                sprintf(sf_wcl, "%sibm%02d/ibm%02d.wcl", TEST_DIR, f_no, f_no);

                f_nodes = fopen(sf_nodes, "r");
                f_nets = fopen(sf_nets, "r");
                f_pl = fopen(sf_pl, "r");
                f_scl = fopen(sf_scl, "r");
                f_wcl = fopen(sf_wcl, "r");

#ifdef DEBUG_VB_ALGO
                fprintf(stderr, "Opened Files:\n\t%s\n\t%s\n\t%s\n\t%s\n\t%s\n", sf_nodes, sf_nets, sf_pl,
sf_scl, sf_wcl);
#endif
                /*********************** Allocate Memory
*******************************************************/
                //read from files and generate initial data structures based on info
                GenerateNodes(f_nodes);
                GenerateNetlist(f_nets);
                GeneratePlacement(f_pl);
                GenerateGrid(f_scl);
                InitOverlapGrid();
                InitGrid();
                InitEmptyNodeList(Modules*9);

                //wcl only contains area informaition
#ifdef DEBUG
                fprintf(stderr, "Height %d\t Width %d\tModules %d\tPadOffset %d\n", GridHdr[NumRows]-
>coordinate, RowWidth, Modules, PadOffset);
#endif
                gettimeofday(&tv_init, NULL);
#ifdef DEBUG
                 fprintf(stderr, "Creating initial placement \n");
#endif
                FillOverlapGrid();
                cost_org = 0;
                for(i=0; i<Modules; i++)
                 {
                        n = N_Arr[i];
                        if(n==NULL){continue;}
                        cost_org += n->cost;
                 }


#ifdef DEBUG
                fprintf(stderr, "Creating better placement \n");
#endif
                sa_algorithm_overlap(Modules); //do algorithm
                //FillGrid(); //get rid of overlaps
                //sa_algorithm_shift(Modules, INIT_TEMPERATURE);
#ifdef DEBUG
                fprintf(stderr, "Analyzing Results\n");
#endif
                cost = 0;
                for(i=0; i<Modules; i++)
                {
                        n = N_Arr[i];
                        if(n==NULL){continue;}
                        cost += n->cost;
                }
```

```
	gettimeofday(&tv_done, NULL);

	//report
#ifdef OUTPUT_CSV
	fprintf(stdout, "%d,%d,%ld\n", cost, cost_org - cost, (tv_done.tv_sec - tv_start.tv_sec)*S2US +
tv_done.tv_usec - tv_start.tv_usec);
#else
	 fprintf(stdout, "Cost: %d\nImprovement: %d\nTime: %ld\n", cost, cost_org - cost,
(tv_done.tv_sec - tv_start.tv_sec)*S2US + tv_done.tv_usec - tv_start.tv_usec);
#endif

close_files:

exit:
	return return_code;

}

void sa_algorithm_no_overlap(int iterations)
{
	struct node* n;
	struct edge * e;
	struct hyperedge* h;
	double temperature, alpha;
	double acceptance;
	int i, nindex;
	int org_cost, cost_mod;

	alpha = INIT_ALPHA;
	for(temperature = INIT_TEMPERATURE; temperature > FINAL_TEMPERATURE; temperature*=alpha)
	{
		for(i=0; i<iterations; i++)
		{
			nindex = rand()%(Modules-PadOffset);
			n = N_Arr[nindex];
			//org_cost = GetOverlapNodeCost(n);
			cost_mod = MoveRandom(n);
			//cost_mod = GetOverlapNodeCost() - org_cost;
			if(cost_mod < 0){ acceptance = exp(-cost_mod/temperature) * RAND_PRECISION;}
			else{acceptance = 0; } //guarenteed acceptance
			if(rand()%RAND_PRECISION > acceptance)
			{
				AcceptOverlapMove();
			}
			else
			{
				RejectOverlapMove();
			}
		}
		update_iterations(&iterations);
		update_alpha(&alpha);
	}
}


void sa_algorithm_shift(int iterations)
{
	struct node* n;
	struct edge * e;
	struct hyperedge* h;
	double temperature, alpha;
	double acceptance;
	int i, nindex;
	int org_cost, cost_mod;

	alpha = INIT_ALPHA;
	for(temperature = INIT_TEMPERATURE; temperature > FINAL_TEMPERATURE; temperature*=alpha)
	{
		for(i=0; i<iterations; i++)
```

```
                {
                        nindex = rand()%(Modules-PadOffset);
                        n = N_Arr[nindex];
                        //org_cost = GetOverlapNodeCost(n);
                        cost_mod = MoveShift(n);
                        //cost_mod = GetOverlapNodeCost() - org_cost;
                        if(cost_mod < 0){ acceptance = exp(-cost_mod/temperature) * RAND_PRECISION;}
                        else{acceptance = 0; } //guarenteed acceptance
                        if(rand()%RAND_PRECISION > acceptance)
                        {
                                AcceptOverlapMove();
                        }
                        else
                        {
                                RejectOverlapMove();
                        }
                }
                update_iterations(&iterations);
                update_alpha(&alpha);
        }
}


void sa_algorithm_overlap(int iterations)
{
        struct node* n;
        struct edge * e;
        struct hyperedge* h;
        double temperature, alpha;
        double acceptance;
        int i, nindex;
        int org_cost, cost_mod;

        alpha = INIT_ALPHA;
        for(temperature = INIT_TEMPERATURE; temperature > FINAL_TEMPERATURE; temperature*=alpha)
        {
                for(i=0; i<iterations; i++)
                {
                        nindex = rand()%(Modules-PadOffset);
                        n = N_Arr[nindex];
                        cost_mod = MoveOverlapRandom(n);
                        if(cost_mod < 0){ acceptance = exp(-cost_mod/temperature) * RAND_PRECISION;}
                        else{acceptance = 0; } //guarenteed acceptance
                        if(rand()%RAND_PRECISION > acceptance)
                        {
                                AcceptOverlapMove();
                        }
                        else
                        {
                                RejectOverlapMove();
                        }
                }
                update_iterations(&iterations);
                update_alpha(&alpha);
        }
}

void update_iterations(int* iter)
{
        return;
}
void update_alpha(double *alpha)
{
        static increase=1;
        if(*alpha>=0.95){increase = 0;}
        if(increase==1){*alpha += 0.01;}
        else{*alpha -= 0.01; }
        return;
}
```

generate_data.h

```
int GenerateNodes(FILE* gno_file);
int GenerateNetlist(FILE* gnt_file);
int GeneratePlacement(FILE* gpl_file);
int GenerateGrid(FILE* gg_file);
void PopulateCopy();
```

generate_data.c

```
#include <stdio.h>
#include  <fcntl.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <limits.h>
#include <assert.h>
#include"parameters.h"
#include "generate_data.h"
#include "grid.h"
#include "overlapgrid.h"
#include "node.h"
#include "helper.h"

#define NUMROW_PAD 1
#define OVERLAP_NUMROW_PAD 2
#define ROW_PAD 2
#define OVERLAP_ROW_PAD 4

#ifdef DEBUG_VB_GEN
// #define DEBUG_VB_GENNODES
// #define DEBUG_VB_GENNETS
// #define DEBUG_VB_GENPL
 #define DEBUG_VB_GENGRID
#endif
/************************** Input Processing/Initial Netlist generation
*********************************/
int GenerateNodes(FILE* gno_file)
{
	struct node *n, *n_cpy;
	char type;
	int index, index_mod;

	Endline(gno_file);


	//fnodes - need to get number of terminals(pads) and number of total nodes (modules)
	Modules = GetNextInt(gno_file); //number of nodes
	PadOffset = Modules - GetNextInt(gno_file); //terminal offset

#ifdef DEBUG_VB_GEN
	fprintf(stderr, "Reading Nodes\nModules %d PadOffset %d\n", Modules, PadOffset);
#endif


	do{N_Arr = malloc((Modules+1)*sizeof(struct node*));} while(N_Arr==NULL);
	do{N_ArrCpy = malloc((Modules+1)*sizeof(struct node*));} while(N_ArrCpy==NULL);
	for(index = 0; index<=Modules; index++)
	{
		N_Arr[index] = NULL;
		N_ArrCpy[index]=NULL;
	}
	while( (type = (char) fgetc(gno_file))!=EOF)
	{

		do
		{
```

```
                        type = (char) fgetc(gno_file);
                        if(type==EOF){return 1;}
                        if(type == '#'){ Endline(gno_file);continue;}
                }while(!isalpha(type));

                index_mod = index = GetNextInt(gno_file);
                index_mod += (type=='p')?PadOffset:0;

                do{ n = (struct node*) malloc(sizeof(struct node));} while(n==NULL);
                do{ n_cpy = (struct node*)malloc(sizeof(struct node));}while(n_cpy==NULL);
#ifdef DEBUG_VB_GENNODES
                fprintf(stderr, "Creating Node %c%d\n", type, index);
#endif
                n->type = type;
                n->index = index;
                n->locked = 0;

                n->cost = INT_MAX;
                n->birth = NULL;
                n->out_head = NULL;
                n->dir = '\0';
                n->orientation = '\0';

                n->x  = INIT_X;
                n->y = INIT_Y;
                n->width = GetNextInt(gno_file);
                n->height = GetNextInt(gno_file);

                n->north = NULL;
                n->south = NULL;
                n->east = NULL;
                n->west = NULL;
#ifdef DEBUG_VB_GENNODES
                fprintf(stderr, "Created Node %c%d: %d\n", type, index, index_mod);
#endif

                assert(N_Arr[index_mod]==NULL);
                N_Arr[index_mod] = n;
                CopyNode(n, n_cpy);
                N_ArrCpy[index_mod] = n_cpy;
                Endline(gno_file);
        }
}

int GenerateNetlist(FILE* gnt_file)
{
        struct node *n, *n_head;
        struct edge* e, *e_fost, *e_net;
        struct hyperedge * h, *h_net;

        char type,  direction;
        int i, index_mod, node_degree;
        char o_xflag, b_flag, p_flag; //output exclusive flag(only first node in set should be an
output, unless pad), bidirectional flag, pad flag
        int nets, pins;

        Endline(gnt_file);
        //fnets - neet to genertate the netlist graph
        nets = GetNextInt(gnt_file);
        pins = GetNextInt(gnt_file);

        while((node_degree = GetNextInt(gnt_file))!= INT_MIN)
        {

#ifdef DEBUG_VB_GENNETS
                fprintf(stderr, "New Net\n");
#endif

                o_xflag = 1; p_flag = 0; b_flag = 0;
```

```
		direction = '\0';
		Endline(gnt_file);
		for(i=0; i<node_degree; i++)
		{
			//Get information
			do
			{
				type = (char) fgetc(gnt_file);
				if(type==EOF){return 1;}
				if(type == '#'){ Endline(gnt_file);continue;}
			}while(!isalpha(type));
			index_mod = GetNextInt(gnt_file);
			if(type=='p')
			{
				index_mod += PadOffset;
				p_flag = 1;
			}
			do{ direction = (char) fgetc(gnt_file);} while(!isalpha(direction));
			direction =  toupper(direction);
			if(direction == 'O'){o_xflag = (i==0)?1:0;} //only first in list should be output
			if(direction == 'B'){b_flag = 1;}

			//fetch/create node
			if((n = N_Arr[index_mod])==NULL){fprintf(stderr, "NULL node unexpected");}
			if(type == 'p'){n->dir = direction;}
#ifdef DEBUG_VB_GENNETS
			fprintf(stderr, "\t%c: %d %c\n", type, index_mod, direction);
#endif
			if(i==0)
			{
				n_head = n; //new start

				//new output hyperedge - link to parent node
				do{ h_net = malloc(sizeof(struct hyperedge));} while(h_net==NULL);
				h_net->out = n_head;
				h_net->out_head = NULL;
				h_net->next == NULL;

				//insert output hyperedge
				h = n_head->out_head;
				if(h==NULL)
				{
					n_head->out_head = h_net;
				}
				else
				{
					while(h->next!=NULL){ h=h->next;} //advance to end of linked list
					h->next = h_net;
				}

				//cleanup
				Endline(gnt_file);
				n = NULL;
				e = e_net = NULL;
				h = NULL; //we are done - reset

				//OTHER NODES
			}else
			{
				//create edge - link to parent edge
				do{ e =  malloc(sizeof(struct edge));} while(e==NULL);
				e->in = n;
				e->next = NULL;
				e->foster = NULL;
				e->parent = h_net;

				//set edge as input to node n
				e_fost = n->birth;
				if(e_fost == NULL)
```

```
                        {
                                n->birth = e; //first input to this node (a new node)
                        }
                        else
                        {
                                while(e_fost->foster != NULL)
                                {
                                        e_fost = e_fost->foster; //progress to tail
                                }
                                e_fost->foster = e; //add input to tail of linked list
                        }

                        //link hypredge -> edge
                        h = h_net; //load hyperedge
                        e_net = h->out_head;
                        if(e_net==NULL) //first edge - start new inked list
                        {
                                h->out_head = e; //first edge
                        }
                        else //move to tail and insert
                        {
                                while(e_net->next!=NULL){ e_net = e_net->next; }
                                e_net->next = e;
                        }

                        //cleanup
                        Endline(gnt_file);
                        n = NULL;
                        h = NULL;
                        e = NULL;

                }//end else
        }//end for node_degree
        if(p_flag)
        {
                assert(node_degree==2);
        }
        else //no pin in net
        {
                assert(o_xflag); //need one output
                assert(!b_flag); //no bidirectional pins
        }
    }//end while
    return 1;
}//end get input function

int GeneratePlacement(FILE* gpl_file)
{
    struct node* n;
    char type;
    int index_mod;
#ifdef DEBUG_VB_GEN
    fprintf(stderr, "Reading Placements\n");
#endif

    Endline(gpl_file); //get rid of header
    while(1)
    {
        do
        {
                type = (char) fgetc(gpl_file);

                if(type==EOF){return 1;}
                if(type == '#'){ Endline(gpl_file);continue;}

        }while(!isalpha(type));
        index_mod = GetNextInt(gpl_file);
        index_mod += (type=='p')?PadOffset:0;
```

```
                n = N_Arr[index_mod];
                assert(n!=NULL);

                n->x = GetNextInt(gpl_file);
                n->y = GetNextInt(gpl_file);
                n->orientation = GetOrientation(gpl_file);
#ifdef DEBUG_VB_GENPL
                if(/*n->x !=0 || n->y !=0*/1){ fprintf(stderr, "Placed %c%d (%5d,%5d)\n", n->type, n-
>index, n->x, n->y);}
#endif
                if(Endline(gpl_file) ==-1){ break; }
        }
}

int GenerateGrid(FILE* gg_file)
{
        struct node **np, **npcpy; //columns of grid
        struct overlap_node **onp, **onpcpy; //columns of overlap grid
        struct grid_hdr* rhdr; //information about the row
        int  coordinate, height, sitewidth, sitespacing, siteorient, sitesymmetry, subroworgin,
numsites;
        int i, j;
#ifdef DEBUG_VB_GEN
        fprintf(stderr, "Allocating Grid Rows\n");
#endif


        Endline(gg_file); //get rid of header

        NumRows = GetNextInt(gg_file);
        do{ GridHdr = (struct grid_hdr**) malloc((OVERLAP_NUMROW_PAD+NumRows) * sizeof(struct grid_hdr*)
+1);} while(GridHdr==NULL);
        do{ Grid = (struct node ***) malloc((NumRows+NUMROW_PAD) * sizeof(struct node**));}
while(Grid==NULL);
        do{ GridCpy = (struct node ***) malloc((NumRows+NUMROW_PAD) * sizeof(struct node**));}
while(Grid==NULL);
        do{ OverlapGrid = (struct overlap_node***) malloc((NumRows+OVERLAP_NUMROW_PAD)*sizeof(struct
overlap_node**));}while(OverlapGrid==NULL);
        do{ OverlapGridCpy = (struct overlap_node***) malloc((NumRows+OVERLAP_NUMROW_PAD)*sizeof(struct
overlap_node**));}while(OverlapGrid==NULL);
#ifdef DEBUG_VB_GEN
        fprintf(stderr, "Reading Grid Rows\n");
#endif

        //make rows
        for(i=0; i<NumRows; i++)
        {
                do{ rhdr = (struct grid_hdr*) malloc(sizeof(struct grid_hdr));} while(rhdr==NULL);
                rhdr->coordinate = GetNextInt(gg_file);
                rhdr->height = GetNextInt(gg_file);
                rhdr-> sitewidth = GetNextInt(gg_file);
                rhdr-> sitespacing = GetNextInt(gg_file);
                rhdr -> siteorient = GetOrientation(gg_file);
                rhdr -> sitesymmetry = GetSymmetry(gg_file);
                rhdr -> subroworgin = GetNextInt(gg_file);
                rhdr -> numsites = GetNextInt(gg_file);
                rhdr -> numindexes = (rhdr->numsites)/GRID_GRAIN + 1;
#ifdef DEBUG_VB_GENGRID
                fprintf(stderr, "Row%3d:%5d\n",i, rhdr->coordinate );
#endif


                GridHdr[i] = rhdr;
                if(i==0){ AvgRowHeight = rhdr->height; RowWidth = (rhdr->numsites)*(rhdr->sitewidth); }
#ifdef DEBUG_VB_GENGRID
                if(i==0)fprintf(stderr, "RowWidth:%5d\n", RowWidth);
#endif
                assert((rhdr->numsites)*(rhdr->sitewidth) == RowWidth); //assum rows are the same width
                //we will allocate an array of linked list for easy access insertion/deletion
```

```
                //indexes will represent a pointer to the start of its range of coordinates

                do{ np = (struct node**) malloc((RowWidth/GRID_GRAIN+ROW_PAD)*sizeof(struct node*));}
while(np==NULL);
                do{ npcpy = (struct node**) malloc((RowWidth/GRID_GRAIN+ROW_PAD)*sizeof(struct node*));}
while(npcpy==NULL);
                Grid[i] = np;
                GridCpy[i] = npcpy;
                for(j=0; j< RowWidth/GRID_GRAIN+ROW_PAD; j++){ Grid[i][j]=NULL; GridCpy[i][j]=NULL; }

                do{ onp = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)
*sizeof(struct overlap_node*));} while(onp==NULL);
                do{ onpcpy = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)
*sizeof(struct overlap_node*));} while(onpcpy==NULL);
                OverlapGrid[i] = onp;
                OverlapGridCpy[i] = onpcpy;
                for(j=0; j<RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD; j++){ OverlapGrid[i][j]=NULL;
OverlapGridCpy[i][j]=NULL; }

                //cleanup
                np = NULL;
                rhdr = NULL;
                Endline(gg_file);
        }
        //pad row 1
        do{ rhdr = (struct grid_hdr*) malloc(sizeof(struct grid_hdr));} while(rhdr==NULL);
        rhdr->coordinate = GridHdr[i-1]->coordinate + GridHdr[i-1]->height;
        GridHdr[i] = rhdr;

        do{ np = (struct node**) malloc((RowWidth/GRID_GRAIN+ROW_PAD)*sizeof(struct node*));}
while(np==NULL);
        do{ npcpy = (struct node**) malloc((RowWidth/GRID_GRAIN+ROW_PAD)*sizeof(struct node*));}
while(npcpy==NULL);
        Grid[i] = np;
        GridCpy[i] = npcpy;
        for(j=0; j< RowWidth/GRID_GRAIN+ROW_PAD; j++){ Grid[i][j]=NULL; GridCpy[i][j]=NULL; }

        do{ onp = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)  *sizeof(struct
overlap_node*));} while(onp==NULL);
        do{ onpcpy = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)
*sizeof(struct overlap_node*));} while(onpcpy==NULL);
        OverlapGrid[i] = onp;
        OverlapGridCpy[i] = onpcpy;
        for(j=0; j<RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD; j++){ OverlapGrid[i][j]=NULL; OverlapGridCpy[i]
[j]=NULL; }

        i++;
        //pad row 2 (Overlap only)
        do{ rhdr = (struct grid_hdr*) malloc(sizeof(struct grid_hdr));} while(rhdr==NULL);
        rhdr->coordinate = INT_MIN;
        GridHdr[i] = rhdr;
        AvgRowHeight = GridHdr[NumRows]->coordinate/NumRows;

         do{ onp = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)  *sizeof(struct
overlap_node*));} while(onp==NULL);
         do{ onpcpy = (struct overlap_node**) malloc((RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD)
*sizeof(struct overlap_node*));} while(onpcpy==NULL);
         OverlapGrid[i] = onp;
         OverlapGridCpy[i] = onpcpy;
         for(j=0; j<RowWidth/GRID_GRAIN+OVERLAP_ROW_PAD; j++){ OverlapGrid[i][j]=NULL; OverlapGridCpy[i]
[j]=NULL; }
        i++;

        assert(i == NumRows + OVERLAP_NUMROW_PAD);
        return 0;
}

void PopulateCopy()
{
        struct node *n, *ncpy;
```

```
        int i;
        for(i=0; i<Modules; i++)
        {
                n = N_Arr[i];
                ncpy = N_ArrCpy[i];
                if(n==NULL){continue;}
                assert(ncpy!=NULL);
                CopyNode(n, ncpy);
        }
}
```

grid.h

```
struct node *** Grid, ***GridCpy;
struct node** EmptyNodeList, **EmptyNodeListCpy;

void InitGrid(); //initializes several important variables- does not allocate grid
void FillGrid(); //fills the grid with the nodes in N_Arr
void InitEmptyNodeList(int size);//initializes the empty node list
struct node* CreateEmptyNode(int x, int y, int height, int width); //Creates an empty node - does not
insert the node into the grid
struct node * InsertNode(struct node* insert, int x, int y, int* err); //insert a node into the grid
int MoveRandom(struct node* move); //moves a node to a random position
int MoveLocal(struct node* move, struct node* keepout); //moves a node to one of the boundries of
keepout; recursive when nodes block
void AcceptMove(); //copies data structres to  backup copy
void RejectMove(); //restores data structures by copying from backup copy
```

grid.c

```
#include <stdio.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <assert.h>
#include"parameters.h"
#include "grid.h"
#include "node.h"

//variables unique to this module
static char gridLock;
int emptyNodeID, emptyNodeIDCpy;
int eNodeListSize;
int emptyNodeCnt, emptyNodeCntCpy;
//local functions
int get_new_empty_node_id();
void listremove_empty_node(struct node* remove);
void listremove_empty_nodecpy(struct node* remove);
void remove_node(struct node* remove);
struct node* combine_ew(struct node* east, struct node* west, struct node* filler);
struct node* create_filler_node(struct node* copy);
struct node* create_expansion_node(struct node* copy);
struct node* find(int x, int y);
void update_grid(struct node* update, struct node* replace);
void update_all_boundries(struct node* n);
void update_boundry(struct node* center, char orient);

void InitGrid()
{
	gridLock = -1;
}
void FillGrid()
{
	struct node *n, *blocking;
	int i, err;
	gridLock = 0;
	//first make entire grid empty
	n = CreateEmptyNode(0,0,GridHdr[NumRows]->coordinate, RowWidth); //create and empty Node
	n->north = n->south = n->east = n->west = NULL;
	update_grid(n, n);

	for(i=Modules - PadOffset; i>=0; i--)
	{
		n = N_Arr[i];
		if(n==NULL){ continue;}
		while((blocking=InsertNode(n, n->x, n->y, &err))!=NULL)
		{
			if(err !=0){n->x--;}
			MoveLocal(blocking, n);
		}
	}

	AcceptMove();//copy to working copy
```

```
}
void InitEmptyNodeList(int size)
{
	int i;
	eNodeListSize = size;
	do{ EmptyNodeList = (struct node**) malloc(eNodeListSize * sizeof(struct node*)); }
while(EmptyNodeList==NULL);
	do{ EmptyNodeListCpy = (struct node**) malloc(eNodeListSize * sizeof(struct node*)); }
while(EmptyNodeList==NULL);
	for(i=0; i<= size; i++)
	{
		EmptyNodeList[i] = NULL;
	}
	emptyNodeID = 0;
	emptyNodeCnt = 0;
}
int get_new_empty_node_id()
{
	while(EmptyNodeList[emptyNodeID]!=NULL){emptyNodeID++; }
	emptyNodeCnt = (emptyNodeID>emptyNodeCnt)?emptyNodeID:emptyNodeCnt;
	assert(eNodeListSize>emptyNodeCnt);
	return emptyNodeID;
}

struct node* CreateEmptyNode(int x, int y, int height, int width)
{
	struct node* n, *move;

	do{n = (struct node*) malloc(sizeof(struct node));} while(n==NULL); //create node

	n->type  = 'e'; //empty
	n->index = get_new_empty_node_id(); //not used for now
	n->locked = 0;
	n->cost = 0;
	n->birth = NULL;
	n->out_head = NULL;
	n->orientation = OR_N;

	n->x = x;
	n->y = y;
	n->width = width;
	n->height = height;

	EmptyNodeList[n->index] = n;
	return n;
}
void listremove_empty_node(struct node* remove)
{
	 int index;
	 assert(remove!=NULL);
	 emptyNodeIDCpy = remove->index;
	 EmptyNodeList[remove->index] = NULL;
	 free(remove);
}

void listremove_empty_nodecpy(struct node* remove)
{
	int index;
	assert(remove!=NULL);
	emptyNodeIDCpy = remove->index;
	EmptyNodeListCpy[remove->index] = NULL;
	free(remove);
}
/*************************************************************
* This function will insert a node in an empty spot
* If another node (non empty) node is present in that spot, it will return a pointer to
* the node in the way
* Upon success, a NULL pointer is returned and err =0
*************************************************************/
struct node * InsertNode(struct node* insert, int x, int y, int* err)
{
	struct node* replace, *expand, *new_node;
	struct node *north, *south, *east, *west;
	int gindexy, gindexy_original, gindexy_stop;
```

```
        int xorg, yorg;
        int i;

        xorg = insert->x;
        yorg = insert->y;
        //get to starting row
        gindexy = y/AvgRowHeight;
        while(GridHdr[gindexy]->coordinate > y){gindexy--;}
        while(GridHdr[gindexy]->coordinate <= y){gindexy++;}
        gindexy_original = gindexy;
        y= GridHdr[gindexy]->coordinate; //allign y to start of row

        //test if space is free

        replace = find(x, GridHdr[gindexy]->coordinate);
        do
        {
                assert(replace!=NULL);
                if(replace ->type != 'e'){ return replace;}
                if(replace->east==NULL)
                {
                        if(replace->x + replace->width < x + insert->width) //boundry reached
                        {
                                *err = -1; //error occurred
                                return NULL;
                        }
                }
                else if(replace->east->x < x + insert->width) //east is non empty node that overlaps with
insert
                {       assert(replace->east->type != 'e');
                        return replace->east;
                }
                //advance ot next node
                replace = find(replace->x, replace->y+replace->height);
                 if(replace == NULL){ break; }
                 while(GridHdr[gindexy+1]->coordinate < replace->y){ gindexy++; }
        }while(GridHdr[gindexy]->coordinate <= y + insert->height);
        gindexy_stop = gindexy;

        /*********************************************************************
        * If we made it this far, we passed the space-empty test and can start inserting the node
        *********************************************************************/

        //divide north/south boundries of place we are inserting the node
         for(i=0; i<2; i++)
        {
                gindexy = (i==0)?gindexy_original:gindexy_stop-1; //north boundry or south boundry. based
on iteration
                replace = find(x, GridHdr[gindexy]->coordinate);
                 if((i==0 && replace->y < y) || //north boundry divide
                        (i==1 && replace->y + replace->height  > y + insert->height)) //south boundry
divide
                {
                        //insert will be north most
                        east = find(replace->x + replace->width, y+insert->height); //replace->east
                        west = find(replace->x - 1, y + insert->height - 1); //new_empty_node->west;
new_empty_node will be called north
                         north = CreateEmptyNode(replace->x, replace->y, replace->width, y + insert-
>height - replace->y); //replace->north

                        gridLock++;
                        north->north = replace->north;
                        north->east = replace->east;
                        north->south = replace;
                        north->west = west;

                        replace->height = replace->height - north->height;//shrink
                        replace->north = north;
                        replace->east = east;

                        update_grid(east, east);
                        update_all_boundries(east);
                        //update_grid(replace); //dont need since we are just shrinking replace and
updated new+empty_nodes ptrs already
```

```
                        //update_all_boundries(replace);
                        gridLock--;
                 }
        }

        //lets overlay insert into the grid
        //have insert point to other nodes, but Grid has no clue he is there
        insert->x = x;
        insert->y = y;
        insert->north = find(x + insert->width -1, y-1);
        insert->east = find(x+insert->width, y);
        insert->south = find(x, y + insert->height);
        insert->west = find(x-1, y+insert->height-1);
        //begin expansion node, which will maintain grid structure as we make way for insert
        expand = create_expansion_node(insert);
        gindexy=gindexy_original;
        replace = find(x, GridHdr[gindexy]->coordinate);
        while(GridHdr[gindexy]->coordinate <= y + insert->height)
        {
                 assert(replace!=NULL);
                if(replace->x == x && replace->width == insert->width) //same x and width
                {
                        gridLock++;
                        expand->height += replace->height;
                        expand->south = replace->south;
                        expand->west = replace->west;
                        update_grid(expand, expand);
                        update_all_boundries(expand);
                        listremove_empty_node(replace);
                        gridLock--;
                }
                else if(replace->x == x) //same start, but empty node extends further
                {
                        //shrink replace width from west to east
                        south = find(x + insert->width, replace->y + replace->height); //replace south
                        gridLock++;
                        //expand expand
                        expand->height += replace->height;
                        expand->south = replace->south;
                        expand->west = replace->west;

                        replace->x = x + insert->width;
                        replace->width = replace->width - insert->width;
                        replace->south = south;
                        replace->west = expand;

                        update_grid(replace, replace);
                          update_all_boundries(replace);
                        gridLock--;
                }
                else if(replace->x + replace->width == x + insert->width)  //share same west boundry
                {
                        north = find(x -1, replace->y-1); //replace north
                        south = find(x, replace->y+replace->height); //expand->south

                        gridLock++;
                        expand->height += replace->height;
                          expand->south = south;
                          expand->west = replace;

                        replace->width = x - replace->x; //shrink
                        replace->north = north;
                        replace->east = expand;

                          update_grid(replace, replace);
                          update_all_boundries(replace);

                        gridLock--;
                }
                else //free space to the east and west of insert
                {
                        //subdivide replace
                        new_node = CreateEmptyNode(x+insert->width, y, replace->x + replace->width - x -
insert->width, replace->height);
```

```
                        InsertNode(new_node, new_node->x, new_node->y, &i); //easiest way to insert the
node
                        continue; //with replace node subdivided, we can insert as in the above else_if
case
                }

                //move to next node
                replace = find(replace->x, replace->y+replace->height);
                if(replace == NULL){ break; }
                while(GridHdr[gindexy+1]->coordinate < replace->y){ gindexy++; }
        }

        //replace insert with expand
        assert(expand->height == insert->height); //other expand parameters were never changed
        gridLock++;
        assert(insert->north == expand->north);
        insert->east = expand->east; //can be different in the case a new node was inserted
        assert(insert->south == expand->south);
        assert(insert->west == expand->west); //same; new nodes only inserted to the east

        update_grid(insert, insert);
        update_all_boundries(insert);
        gridLock--;
        *err = Cost(insert, xorg, yorg);
        return NULL;

}

/*****************************************************
* Only empty nodes are freed from memory
* Other nodes are inserted on the Remove Stack to be reinserted into the grid
*****************************************************/
void remove_node(struct node* remove)
{
        int i, j;
        struct node* nearby, *east, *west, *filler;

        assert(remove->type != 'f' && remove->type != 'F');
        //combine sorrrounding empty nodes, unlink pointers to remove node (recursive)
        filler = create_filler_node(remove);
        combine_ew(remove->east, remove->west, filler);
        assert(filler->height == 0);
        //theres a chance the upper replacement node is linked to filler->north
        if(filler->south->north == filler){filler->south->north = filler->north;}
        free(filler);

        //unlink node from grid
        remove->north = NULL;
        remove->east = NULL;
        remove->west = NULL;
        remove->south=NULL;
        remove->x = INIT_X;
        remove->y = INIT_Y;

        //next store in removed node stack
        if(remove->type == 'e'){ listremove_empty_node(remove);}

}

/*****************************************************************************************
 * This recursive method must take advantage of the fact that east comes from the north, and west comes
from the south
 * We will recursively move east down southward until it extends it is more south than the west node
 * Then we will combine the nodes
 * We return the west node that is in line with the latitude we are operating at
 *****************************************************************************************/
struct node* combine_ew(struct node* east, struct node* west, struct node* filler)
{
        struct node *retnode, *tnorth, *tsouth, *teast, *twest; //return node, and temporary direction
nodes
        int iheight;

        assert((east->y + east->height) <= (west->y + west->height)); //should not go to far south
        if((east->y + east->height) != (west->y + west->height))
```

```
	{
		west = combine_ew(east->south, west, filler);
	}
	assert(west!=NULL);

	if(east->type=='e' && west->type=='e') //empty
	{
		assert(east->y +east->height == west->y + west->height); //should have same terminating
line
		if(east->y == west->y)
		{
			retnode = west->north;
			twest = west->north; //find(west->x-1, west->y-1); //filler->west
			gridLock++;
			//expand west eastward
			west->width = west->width + filler->width + east->width; //extend
			west->north = east->north; //fix pointers for west node
			west->east = east->east;
			//remove east
			listremove_empty_node(east);
			//shrink filler northward
			filler->height -= west->height;
			filler->south = west;
			filler->west = twest;

			update_grid(west, west); //fix grid ptrs
			update_all_boundries(west);// fix corner stitching TO west
			gridLock--;

			return retnode; //move up one node
		}
		else if(east->y < west->y)//east higher than west
		{
			retnode = west->north;
			teast = find(east->x + east->width, west->y); //west->east
			gridLock++;
			//shrink east northward
			east->height = west->y - east->y; //shrink
			east->south = west;
			east->west = filler;
			//expand west eastward
			west->width = west->width + filler->width + east->width; //expand
			west->north = east; //fix pointers for west node
			west->east = teast;
			//shrink filler northward
			filler->height -= west->height;
			filler->south = west;
			filler->west = west->north;

			update_grid(west, west);  //fix grid ptrs
			update_all_boundries(west);// fix corner stitching TO west
			gridLock--;
			return combine_ew(east, retnode, filler); //need to perform operation on east
again
		}
		else //if(east->y > west->y) //west higher than east
		{
			twest = find(west->x-1, east->y-1); //west->west
			gridLock++;
			//expand east westward
			east->x = west->x; //shift
			east->width = west->width + filler->width + east->width; //expand
			east->south = west->south; //fix pointers from east node
			east->west = west->west;
			//shrink west northward
			west->height = east->y - west->y; //shrink
			west->south = east;
			west->west = twest;
			//shrink filler northward
			filler->height -= east->height;
			filler->south = east;
			filler->west = west;

			update_grid(east, east); //fix grid ptrs
```

```
                        update_all_boundries(east);// fix corner stitching TO east
                        gridLock--;
                        return west;
                }
        }
        else if(east->type == 'e') //west is actual node
        {
                if(east->y == west->y)
                {
                        gridLock++;
                        //expand east westward
                        east->x = filler->x; //shift
                        east->width += filler->width; //extend east
                        east->west = west;
                        east->south = filler->south; //find south (SW corner)
                        //shrink filler northward
                        filler -= east->height;
                        filler->west = west->north;
                        filler->south = east;

                        update_grid(east, east); //fix grid ptrs
                        update_all_boundries(east);// fix corner stitching TO east
                        gridLock--;
                        return west->north; //move up one node
                }
                else if(east->y < west->y)//east higher than west
                {
                        //create new empty node to fill in node space
                        assert(east->height + east->y <= west->height+west->y);
                        /*iheight = (east->y + east->height < west->y + west->height) ?
                                east->y + east->height - west->y :
                                west->height;*/
                        tsouth = CreateEmptyNode(filler->x, west->y, filler->width + east->width, east->y
+ east->height - west->y); //filler->south
                        teast = find(east->x+east->width, west->y); //tsouth->east
                        gridLock++;
                        //fix new_node (tsouth) pointers
                        tsouth->north = east;
                        tsouth->east = teast;
                        tsouth->south = filler->south;
                        tsouth->west = west;
                        //shrink east northward
                        east->height -= tsouth->height;
                        east->south = tsouth;
                        east->west = filler;
                        //shrink filler
                        filler->height -= tsouth->height;
                        filler->south = tsouth;
                        filler->west = west->north;

                        update_grid(tsouth, tsouth);
                        update_all_boundries(tsouth);
                        gridLock--;
                        return combine_ew(east, west->north, filler); //need to perform operation on east
again
                }
                else// if(east->y > west->y) //west higher than east
                {

                        gridLock++;
                        //expand east westward
                        east->x = filler->x; //shift
                        east->width = filler->width + east->width; //expand
                        east->south = filler->south;
                        east->west = west;
                        //shrink filler northward
                        filler->height -= east->height;
                        filler->south = east;
                        filler->west = west;

                        update_grid(east, east); //fix grid ptrs
                        update_all_boundries(east);// fix corner stitching TO east
                        gridLock--;
                        return west;
```

```
                }


        }
        else if(west->type == 'e') //east is an actual node
        {
                if(east->y == west->y)
                {
                        retnode = twest = west->north;
                        gridLock++;
                        //expand west eastward
                        west->width = west->width + filler->width; //extend east
                        west->north = filler;
                        west->east = east;
                        //shrink filler northward
                        filler->height -= west->height;
                        filler->south = west;
                        filler->west = twest;

                        update_grid(west, west); //fix grid ptrs
                        update_all_boundries(west);
                        gridLock--;
                        return retnode;
                }
                else if(east->y < west->y) //east is higher than west
                {
                        retnode = twest = west->north;
                        gridLock++;
                        //expand west
                        west->width = west->width + filler->width;
                        west->north = filler;
                        west->east = east;
                        //shrink filler
                        filler->height -= west->height;
                        filler->south = west;
                        filler->west = twest;

                        update_grid(west, west); //fix grid ptrs
                        update_all_boundries(west);
                        gridLock--;
                        return combine_ew(east, retnode, filler); //still need work on east, with new
west
                }
                else //if (east->y > west->y) //west higher than east
                {
                        iheight = west->y + west->height - east->y;
                        //subdivide west; new node to be exrended eastward so that it is south of filler
                        tsouth = CreateEmptyNode(west->x, east->y, west->width + filler->width, iheight);
//filer->south
                        twest = find(west->x -1, tsouth->y-1);
                        gridLock++;
                        //link in tsouth
                        tsouth->north = filler;
                        tsouth->east = east;
                        tsouth->south = west->south;
                        tsouth->west = west->west;
                        //shrink west northward
                        west->height -= tsouth->height;
                        west->south = tsouth;
                        west->west = twest;
                        //shrink filler northward
                        filler->height -= tsouth->height;
                        filler->south = tsouth;
                        filler->west =  west;

                        update_grid(tsouth, tsouth);
                        update_all_boundries(tsouth);
                        gridLock--;
                        return west;
                }

        }
        else //actual nodes on both sides
        {
```

```
		if(east->y == west->y)
		{
			iheight = (east->height < west->height)? east->height : west->height ;
			tsouth = CreateEmptyNode(filler->x, west->y, filler->width, iheight); //filler->south
			gridLock++;
			//link in tsouth
			  tsouth->north = filler;
			  tsouth->east = east;
			  tsouth->south = filler->south;
			  tsouth->west = west;
			  //shrink filler northward
			  filler->height -= tsouth->height;
			  filler->south = tsouth;
			  filler->west = west->north;

			  update_grid(tsouth, tsouth);
			  update_all_boundries(tsouth);
			gridLock--;
			return west->north; //move up one node
		}
		else if(east->y < west->y)//east is higher than west
		{
			iheight = (east->y + east->height < west->y + west->height) ?
				east->y + east->height - west->y :
				west->height;
			retnode = west->north;
			tsouth = CreateEmptyNode(filler->x, west->y, filler->width, iheight ); //filler->south

			gridLock++;
			  //link in tsouth
			  tsouth->north = filler;
			  tsouth->east = east;
			  tsouth->south = filler->south;
			  tsouth->west = filler->west;
			  //shrink filler northward
			  filler->height -= tsouth->height;
			  filler->south = tsouth;
			  filler->west = west->north;

			  update_grid(tsouth, tsouth);
			  update_all_boundries(tsouth);
			gridLock--;
			return combine_ew(east, retnode, filler); //still need work on east, with new west

		}
		else //if(east->y > west->y) //west is higher than east
		{
			iheight = (east->y + east->height < west->y + west->height) ?
				east->height :
				west->y + west->height - east->y ;
			tsouth = CreateEmptyNode(filler->x, east->y, filler->width, iheight ); //filler->south

			gridLock++;
			  //link in tsouth
			  tsouth->north = filler;
			  tsouth->east = east;
			  tsouth->south = filler->south;
			  tsouth->west = filler->west;
			  //shrink filler northward
			  filler->height -= tsouth->height;
			  filler->south = tsouth;
			  filler->west = west;

			  update_grid(tsouth, tsouth);
			  update_all_boundries(tsouth);
			return west;
		}
	}
	return NULL; //error - should not reach here
}
```

```
int MoveRandom(struct node* move)
{
	struct node* blocking;
	int x,y,err, cost;
	cost = 0;
	//remove
	remove_node(move);
	//find random spot
	x = rand()%(RowWidth-move->width);
	do
	{
		y = rand()%NumRows;
	}while(GridHdr[y]->coordinate + move->height > GridHdr[NumRows]->coordinate);
	y = GridHdr[y]->coordinate;

	while((blocking=InsertNode(move, x, y, &err))!=NULL)
	{
		cost += MoveLocal(blocking, move); //move any nodes in the way
	 }
	cost += err;
	return cost;

}
int MoveShift(struct node* move)
{
	 struct node* blocking;
	 int x,y,err, cost;
	 cost = 0;
	 //remove
	 remove_node(move);
	 //shift one spot up or down
	 x = move->x + rand()%3-2;
	if(x != move->x){ y = move->y; }
	 else
	{
		do
		 {
			y = move->y/AvgRowHeight + rand()%5-3;
		 }while(GridHdr[y]->coordinate + move->height > GridHdr[NumRows]->coordinate &&
GridHdr[y]->coordinate != move->y);
		y = GridHdr[y]->coordinate;
	}
	 while((blocking=InsertNode(move, x, y, &err))!=NULL)
	 {
		 cost += MoveLocal(blocking, move); //move any nodes in the way
	 }
	 cost += err;
	 return cost;

}


int MoveLocal(struct node* move, struct node* keepout)
{
	struct node* blocking;
	int err, cost;
	int direction;

	direction = rand()%4;//NESW

	remove_node(move);
	while(1)
	{
		switch(direction)
		{
			case '0': //move north
				while((blocking=InsertNode(move, move->x, keepout->y - move->height,
&err))!=NULL)
				{
					cost += MoveLocal(blocking, move);
				}
			break;
			case '1': //move east
```

```
                                while((blocking=InsertNode(move, keepout->x+keepout->width, move->y,
&err))!=NULL)
                                {
                                        cost += MoveLocal(blocking, move);
                                }
                        break;
                        case '2': //move south
                                while((blocking=InsertNode(move, move->x, keepout->y + keepout->height,
&err))!=NULL)
                                {
                                        cost+=MoveLocal(blocking, move);
                                }
                        break;
                        case '3': //move west
                                while((blocking=InsertNode(move, keepout->x-move->width, move->y, &err))!
=NULL)
                                {
                                        cost += MoveLocal(blocking, move);
                                }
                        break;
                        default:
                                err = -1;
                                break;
                }
                if(err>=0){ break;} //exit from while loop
                //out of bound or default case
                direction = rand()%4;
        }
        cost += err;
        return cost;

}
void AcceptMove()
{
        struct node *n, *ncpy;
        int i, j;
        int ecnt;
        //copy modules
        for(i=0; i<Modules; i++)
        {
                n = N_Arr[i];
                ncpy = N_ArrCpy[i];
                if(n==NULL){continue;}
                assert(ncpy != NULL);
                CopyParallelNode(n, ncpy);
        }
        //copy empty nodes
        ecnt = (emptyNodeCnt>emptyNodeCntCpy)?emptyNodeCnt:emptyNodeCntCpy; //iterate through bogger of
2
        for(i=0; i<ecnt; i++)
         {
                n = EmptyNodeList[i];
                 ncpy = EmptyNodeListCpy[i];
                        if(n==NULL && ncpy!=NULL)
                {listremove_empty_nodecpy(ncpy); }
                else if(n!=NULL && ncpy==NULL)
                {
                        do{ncpy = malloc(sizeof(struct node));}while(ncpy==NULL);
                        CopyParallelNode(n, ncpy);
                        EmptyNodeListCpy[ncpy->index] = ncpy;
                }
                else if(n!=NULL && ncpy!=NULL)
                {CopyParallelNode(n, ncpy);}
         }
        emptyNodeCntCpy = emptyNodeCnt;
        emptyNodeIDCpy = emptyNodeID;
        //copy grid ptrs
        for(i=0; i<NumRows; i++)
        {
                for(j=0; j<RowWidth/GRID_GRAIN; j++)
                {
                        n = Grid[i][j];
                        if(n->type == 'e'){ GridCpy[i][j] = EmptyNodeListCpy[n->index];}
                        else if(n->type == 'a'){ GridCpy[i][j] = N_ArrCpy[n->index];}
```

```
                        else if(n->type == 'p'){ GridCpy[i][j] = N_ArrCpy[n->index+PadOffset];}
                }
        }
}

void RejectMove() //same as accept move except swapped xx<->xxCpy
{
        struct node *n, *ncpy;
        int i, j;
        int ecnt;
        //copy modules
        for(i=0; i<Modules; i++)
        {
                n = N_Arr[i];
                ncpy = N_ArrCpy[i];
                if(ncpy==NULL){continue;}
                assert(n != NULL);
                RestoreParallelNode(ncpy, n);
        }
        //copy empty nodes
        ecnt = (emptyNodeCnt>emptyNodeCntCpy)?emptyNodeCnt:emptyNodeCntCpy; //iterate through bogger of
2
        for(i=0; i<ecnt; i++)
         {
                n = EmptyNodeList[i];
                 ncpy = EmptyNodeListCpy[i];
                        if(ncpy==NULL && n!=NULL)
                {listremove_empty_nodecpy(n); }
                else if(ncpy!=NULL && n==NULL)
                {
                        do{n = malloc(sizeof(struct node));}while(n==NULL);
                        RestoreParallelNode(ncpy, n);
                        EmptyNodeList[n->index] = n;
                }
                else if(ncpy!=NULL && n!=NULL)
                {RestoreParallelNode(ncpy, n);}
         }
        emptyNodeCnt = emptyNodeCntCpy;
        emptyNodeID = emptyNodeIDCpy;
        //copy grid ptrs
        for(i=0; i<NumRows; i++)
        {
                for(j=0; j<RowWidth/GRID_GRAIN; j++)
                {
                        ncpy = GridCpy[i][j];
                        if(ncpy->type == 'e'){ Grid[i][j] = EmptyNodeList[ncpy->index];}
                        else if(ncpy->type == 'a'){ Grid[i][j] = N_Arr[ncpy->index];}
                        else if(ncpy->type == 'p'){ Grid[i][j] = N_Arr[ncpy->index+PadOffset];}
                }
        }

}

struct node* create_filler_node(struct node* copy)
{
        struct node* filler;

        do{filler = (struct node*) malloc(sizeof(struct node));}while(filler==NULL);

        CopyNode(copy, filler);
        filler->type = 'f';
        //replace filler in grid
        update_grid(filler, filler);
        update_all_boundries(filler); //point all nodes to filler
        return filler;
}
struct node* create_expansion_node(struct node* copy)
{
        struct node* expand;

         do{expand = (struct node*) malloc(sizeof(struct node));}while(expand==NULL);

        CopyNode(copy, expand);
        //shrink expand
```

```
        expand->south = find(copy->x, copy->y);
        expand->west = find(copy->x-1, copy->y-1);
         expand->type = 'F';
        expand->height = 0;
         return expand;

}

struct node* find(int x, int y)
{

        int i,j;
        struct node* start, *position;
        assert(gridLock==0);
        if(x<0 || x>=RowWidth || y<0 || y>= GridHdr[NumRows]->coordinate){return NULL;} //out of bounds
        i = y/AvgRowHeight;
         j = x/GRID_GRAIN;
         while(GridHdr[i]->coordinate > y){i--;}; //move to proper row in case of poor avg height
        while(GridHdr[i+1]->coordinate < y){i++;}

        position = Grid[i][j];

        while(1)
        {
                start = position;
                if(position->y > y){ position = position->north;}
                else if(position->y + position->height < y) { position = position->south; }
                if(position->x + position->width < x){ position = position->east;}
                else if(position->x > x){position = position->west;}

                if(start == position){break;}
        }
        return position;
}

void update_grid(struct node* update, struct node* replace)
{
        int i,j;

        assert(update!=NULL && (replace==NULL || replace == update));
        //first update grid ptrs
        i = (update->y)/AvgRowHeight;
        j = (update->x)/GRID_GRAIN;
        while(GridHdr[i]->coordinate > update->y){i--;}; //move to proper row in case of poor avg height

        do
        {
                do
                {
                        if
                                (       update->y <=  GridHdr[i]->coordinate &&
                                        update->y + update->height > GridHdr[i]->coordinate &&
                                        update->x <= j*GRID_GRAIN &&
                                        update->x + update->width > j*GRID_GRAIN
                                )
                                {
                                        Grid[i][j]=replace;
                                }

                        j++;
                }while((j*GRID_GRAIN) < (replace->x + replace->width));
                j = (replace->x)/GRID_GRAIN;
                i++;
        }while(GridHdr[i]->coordinate < (replace->y + replace->height) );

}
void update_all_boundries(struct node* n)
{
        update_boundry(n, OR_N);
        update_boundry(n, OR_S);
        update_boundry(n, OR_E);
        update_boundry(n, OR_W);
}
```

```
void update_boundry(struct node* center, char orient)
{
	struct node* border;
	switch(orient)
	{
		case OR_N: //fix south pointer along center's north border
			border = center->north;
			if(border==NULL){return;}
			while(border->x > center->x)
			{
				assert(border->y + border->height == center->y); //should always be
centers north border
				border->south = center;
				border = border->west;
			}
			break;
		case OR_S:
			border = center->south;
			if(border==NULL){return;}
			while(border->x < center->x + center->width)
			{
				assert(center->y + center->height == border->y); //should always be
centers south border
				border->north = center;
				border = border->east;
			}
			break;
		case OR_E:
			border = center->east;
			if(border==NULL){return;}
			while(border->y < center->y + center->height)
			{
				assert(center->x + center->width == border->x); //should always be centers
east border
				border->west = center;
				border = border->south;
			}
			break;
		case OR_W:
			border = center->west;
			if(border==NULL){return;}
			while(border->y > center->y) //moving north
			{
				assert(border->x + border->width == center->x); //should always be centers
west border
				border->east = center;
				border = border->north;
			}
			break;
		default:
			assert(0);
			break;
	}
}
```

helper.h

```
int LogTwo(int i);
char GetOrientation(FILE* go_file);//gets next orientation from file
int GetNextInt(FILE* gni_file); //gest next integer from file
char GetSymmetry(FILE* gs_file);
int  Endline(FILE* el_file); //finishes reading a line in a file
void Help();
void inline RequestHelp();
```

helper.c

```
#include <stdio.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include  <fcntl.h>
#include <assert.h>
#include <limits.h>
#include "node.h"
#include "helper.h"

//local helper function
int to_integer(char * c);

int LogTwo(int i)
{
        int ret;
        assert(i >= 0);
        for(ret = 0; i !=0; ret++){ i = i>>1; }
        return ret;
}


char GetOrientation(FILE* go_file)
{
       int ret, i;
       char s[2], flipped_xor;
        flipped_xor =0;
         while(1)
        {
                s[0] =(char)fgetc(go_file); //wait for first digit
                switch(s[0])
                {
                        case EOF:
                                return 0;
                        case '#':
                                Endline(go_file);
                                break;//endswtich
                        case 'N':
                        case 'n':
                                return (flipped_xor ^ OR_N);
                        case 'S':
                          case 's':
                                  return (flipped_xor ^ OR_S);
                        case 'E':
                          case 'e':
                                  return (flipped_xor ^ OR_E);
                        case 'W':
                          case 'w':
                                  return (flipped_xor ^ OR_W);
                        case 'F':
                        case 'f':
                                flipped_xor = OR_F;
                                continue; //go back to while (skip resetting the flag below)
                        default:
                                break; //end switch
                }
                flipped_xor = 0; //direction should be immeadiately after next character
```

```
        }
        for(i=1; i<10 && isdigit(s[i] = (char)fgetc(go_file)); i++);
        s[i] = '\0'; //null terminate
        return to_integer(s);
}

int GetNextInt(FILE* gni_file)
{
        int ret, i;
        char s[10];
        char negative_flag;
        //      while(scanf("%d", &ret)<=0); //scanf f stops working after ~400 integer scans
        negative_flag = 0;
        while(!isdigit((s[0]=(char)fgetc(gni_file)))) //wait for first digit
        {
                if(s[0] == '#') { Endline(gni_file); } //remove file
                if(s[0] == '-') { negative_flag = 1; }
                if(s[0] == EOF) { return INT_MIN; }
                else { negative_flag = 0; }
        }
        for(i=1; i<10 && isdigit(s[i] = (char)fgetc(gni_file)); i++);
        s[i] = '\0'; //null terminate
        ret = to_integer(s);
        if(negative_flag==0) return ret;
        else{ return -ret; }
}

char GetSymmetry(FILE* gs_file)
{
        char s;
        while(!isalpha((s=(char)fgetc(gs_file)))) //wait for first digit
        {
                if(s == '#') { Endline(gs_file); } //remove file
        }
        return s;
}

/************* get_line ********************
 * Gets one line from infile
 *
 ************************************************/
int  Endline(FILE* el_file)
{
        unsigned char c[1];
        int err;

        while(1) //avoid end of file and line
        {
                *c = fgetc(el_file);
                if(*c=='\n'){ return 0; }
                if(*c == EOF){ return -1;}
        }

}


/************** to_integer ********************
 * Converts a string to an integer
 *
 * c - numeric string to conver
 *
 * returns the integer
 ********************************************/
int to_integer(char * c)
{
        int i, val;
        i= val = 0;
        while(c[val]!='\0' && c[val]<='9' && c[val]>='0')
```

```
        {
                i+= (unsigned int) c[val] - '0';
                i*=10;
                val++;
        }
        i/=10;
        return i;
}

/************** help ***************
 * prints out help message
 *********************************/
void Help()
{
        fprintf(stdout, "request_help");

}

/***********request_help ***********
 * suggests to confused users that the seek help
 *********************************/

void inline RequestHelp()
{
        fprintf(stdout,  "Use \"./algo -h\" for help");
}
```

image.h

```
void WriteImage(char* image_filename); //writes datastructures to an image file;
```

image.c

```
#include <stdio.h>
#include  <fcntl.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <limits.h>
#include <assert.h>
#include <wand/MagickWand.h>
#include"parameters.h"
#include "node.h"


void writeImage(char* image_filename)
{
        struct node* n;
        int i;
        MagickWand *m_wand = NULL;
        DrawingWand *d_wand = NULL;
        PixelWand *c_wand = NULL;


        m_wand = NewMagickWand();
        d_wand = NewDrawingWand();
        c_wand = NewPixelWand();

        //MagickSetImageFilename(m_wand,  image_filename)

        PixelSetColor(c_wand,"white");
        MagickNewImage(m_wand,RowWidth,GridHdr[NumRows]->coordinate,c_wand);

        DrawSetStrokeOpacity(d_wand,1);

        PushDrawingWand(d_wand);
        PixelSetColor(c_wand,"rgb(0,0,1)");

        DrawSetStrokeColor(d_wand,c_wand);
        DrawSetStrokeWidth(d_wand,0);
        DrawSetStrokeAntialias(d_wand,1);
        for(i=0; i<Modules; i++)
        {
                n = N_Arr[i];
                PixelSetColor(c_wand,"red");
                //DrawSetStrokeOpacity(d_wand,1);
                DrawSetFillColor(d_wand,c_wand);
                DrawRectangle(d_wand,n->x,n->y,n->x+n->width,n->y+n->height);
        }
        PopDrawingWand(d_wand);

        MagickWriteImage(m_wand, image_filename);

        c_wand = DestroyPixelWand(c_wand);
        m_wand = DestroyMagickWand(m_wand);
        d_wand = DestroyDrawingWand(d_wand);

        MagickWandTerminus();
}
```

node.h

```
#ifndef __NODE_H
#define __NODE_H

struct node **N_Arr, **N_ArrCpy;
struct grid_hdr** GridHdr;
int Modules, PadOffset;
int NumRows, AvgRowHeight, RowWidth;


//oreintation characters
#define OR_N 'N'
#define OR_S 'S'
#define OR_E 'E'
#define OR_W 'W'
#define OR_FN 'n'
#define OR_FS 's'
#define OR_FE 'e'
#define OR_FW 'w'
#define OR_F 0x20

struct node
{
	char type; //padi/terminal (p) or cell (a)  or empty (e)
	int index; //pad or cell #
	int locked; //counter to LOCK_THRESH
	int cost;
	struct edge* birth; //first edge input
	struct hyperedge* out_head; //hyperedge output
	char dir; //direction (pads only- other nodes have more than one pin)
	char orientation;

	int x; //x coordinate
	int y; //y coordinate
	int width;
	int height;

	//corner stitching
	struct node* north;
	struct node* south;
	struct node* east;
	struct node* west;

};
struct edge
{
	struct node* in; //the node this hyperedge connects to as an input
	struct hyperedge* parent; //the parent edge of this hyperedge
	struct edge* next; //next out node this net connects to
	struct edge* foster; //next input hyperedge this out node is connected to
};
struct hyperedge
{
	struct node* out; //the node this edge connects to (output signal)
	struct edge* out_head; //head of a linked list of hyperedges this edge expands to
	struct hyperedge* next; //next output the in node is connected to
};
struct overlap_node
{
	struct node* node;    //node
	struct overlap_node * prev; //prev node in this grid sqaure
	struct overlap_node * next; //next node in this grid square
};
struct grid_hdr
{
	int coordinate;
	int height;
```

```
        int sitewidth;
        int sitespacing;
        char siteorient;
        char sitesymmetry;
        int subroworgin;
        int numsites;
        int numindexes;

};


int Cost(struct node* n, int org_x, int org_y); //updates cost of node n and all other surrounding
nodes
void CopyNode(struct node* original, struct node* copy); //copys a nodes information
void CopyParallelNode(struct node* original, struct node* copy); //copys a nodes information from Node
-> NodeCpy
void RestoreParallelNode(struct node* original, struct node* copy); //copys a nodes infromation from
NodeCpy->Node

#endif //__NODE_H
```

node.c

```
#include <stdio.h>
#include  <fcntl.h>
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <assert.h>
#include"parameters.h"
#include "node.h"
#include "helper.h"


/*************************************************
* This function will recalculate the cost of node n and modify the costs of surrounding nodes
* It will return the delta cost for node n
*************************************************/
int Cost(struct node* n, int org_x, int org_y)
{
        int s_in=0, d_in=0; //same partition in, different partition in
        int s_flag, d_flag; //same flag/different flag for chyperedge children
        int cost, cost_mod, cost_original; //cost and modified cost
        struct node* bucket;
        struct edge *e;
        struct hyperedge* h;

        if(n==NULL) return 0;
        cost_original = n->cost;
        cost = 0;
        //inputs
        e = n->birth;
        while(e!=NULL) //iterate through node inputs to determine where parents are
        {
                //calculate difference
                cost_mod =0;
                //x
                cost_mod += abs(e->parent->out->x - org_x) - abs(e->parent->out->x - n->x);
                cost += abs(e->parent->out->x - n->x);
                //y
                cost_mod += abs(e->parent->out->y - org_y) - abs(e->parent->out->y - n->y);
                 cost += abs(e->parent->out->y - n->y);

                e->parent->out->cost -= cost_mod;
                e = e->foster; //move to next input
        }

        //outputs
        h = n->out_head;
        while(h != NULL) //through each  output hyperedge of node
```

```
	{
		e = h->out_head;
		while(e!=NULL) //through each edge of hyperedge
		{
			//calculate difference
			cost_mod =0;
			 //x
			 cost_mod += abs(e->in->x - org_x) - abs(e->in->x - n->x);
				cost += abs(e->in->x - n->x);
			 //y
				cost_mod += abs(e->parent->out->y - org_y) - abs(e->parent->out->y - n-
>y);
			 cost += abs(e->parent->out->y - n->y);

			 e->in->cost -= cost_mod;
			e = e->next;
		}

		h = h->next;
	}
	n->cost = cost;

	return cost_original - cost;
}


void CopyNode(struct node* original, struct node* copy)
{
	assert(original != NULL &&  copy != NULL);

	 copy->type = original->type;
	 copy->index = original->index;
	 copy-> locked =  original ->locked;
	 copy->cost = original->cost;
	 copy->birth = original->birth;
	 copy-> out_head =  original->out_head;
	 copy->dir = original->dir;
	 copy->orientation = original->orientation;

	 copy->x = original->x; //x coordinate
	 copy->y = original->y;
	 copy->width = original->width;
	 copy->height = original->height;

	 //corner stitching
	 copy->north = original->north;
	 copy->south = original->south;
	 copy->east = original->east;
	 copy->west = original->west;

}
void CopyParallelNode(struct node* original, struct node* copy)
{
	int index;
	assert(original->type == 'a');
	assert(original == N_Arr[original->index]); //make sure original is in the main set, not the
copy set

	CopyNode(original, copy);

	 //corner stitching
	 copy->north = N_ArrCpy[original->north->index];
	 copy->south =  N_ArrCpy[original->south->index];
	 copy->east =  N_ArrCpy[original->east->index];
	 copy->west =  N_ArrCpy[original->west->index];

}

void RestoreParallelNode(struct node* original, struct node* copy)
```

```
{
        int index;
        assert(original->type == 'a');
        assert(original == N_ArrCpy[original->index]); //make sure original is in the copy set

        CopyNode(original, copy);

        //corner stitching
        copy->north = N_Arr[original->north->index];
        copy->south =  N_Arr[original->south->index];
        copy->east =  N_Arr[original->east->index];
        copy->west =  N_Arr[original->west->index];

}
```

overlapgrid.h

```
#include "node.h"
//Here is the bin based grid
struct overlap_node *** OverlapGrid, ***OverlapGridCpy;

void InitOverlapGrid(); //initializes dome integers - allocation should be done separately
void FillOverlapGrid();
int getOverlapCost(); //retruns variable
int getExtraRowWidthCost();//returns variable
int InsertOverlapNode(struct node* insert, int x, int y); //inserts a node and recalculates cost
int MoveOverlapRandom(struct node* move); //moves the specified node to a random grid location
void AcceptOverlapMove(); //copies datastructures over to backup cpy
void RejectOverlapMove(); //restores data structures from backup cpy
int CostTimberwolf(struct node* ncost, int xorg, int yorg, int overlap_org, int row_widthorg);
```

overlapgrid.c

```
#include <stdlib.h>
#include<stdint.h>
#include <ctype.h>
#include <assert.h>
#include"parameters.h"
#include "overlapgrid.h"
#include "node.h"

#ifdef DEBUG_VB_OLG //overlap grid
#include<stdio.h>
// #define DEBUG_VB_OLGWO //work overlap
// #define DEBUG_VB_OLGACC //accept
// #define DEBUG_VB_OLGREJ //rejecti
// #define DEBUG_VB_OLGVERIFY
#endif

//variables unique to this module
int gridOverlap, gridOverlapCpy;
int extraRowWidth, extraRowWidthCpy;
void work_overlap(struct node* insert, int insert_flag);
void inline verify();

void InitOverlapGrid()
{
	gridOverlap = gridOverlapCpy = 0;
	extraRowWidth = extraRowWidthCpy = 0;
}
int GetOverlapCost(){return gridOverlap;}
int GetExtraRowWidthCost(){return extraRowWidth;}

void FillOverlapGrid()
{
	int i, x, y;
	struct  node* insert;
	for(i=0; i<=PadOffset; i++)
	{
		insert = N_Arr[i];
		if(insert==NULL){continue;}
		if(insert->x==0 && insert->y==0)
		{
			x = rand()%(RowWidth);//-insert->width);
			y = rand()%(NumRows-1);//-insert->y/AvgRowHeight);
			y = GridHdr[y]->coordinate;
		}
		else
		{
			x = insert->x;
			y = insert->y;
		}
		InsertOverlapNode(insert, x, y);
	}
```

```
        for(i=0; i<Modules; i++)
         {
                 insert = N_Arr[i];
                if(insert==NULL){continue;}
                Cost(insert, 0, 0);
        }
        AcceptOverlapMove();
        return;
}

/*************************************************************
* This function will insert or remove a node into the proper position
* A bound check should be performed before calling this function
*************************************************************/
int InsertOverlapNode(struct node* insert, int x, int y)
{
        int cost, overlap_org, row_cost_org;

        overlap_org = GetOverlapCost();
        row_cost_org = GetExtraRowWidthCost();
         if(y<0){y =0;}

         insert->x = x;
         insert->y = y;
        work_overlap(insert, 1);
        cost  =  CostTimberwolf(insert, x, y,overlap_org, row_cost_org);
        return cost;
}

void work_overlap(struct node* insert, int insert_flag)
{
        struct overlap_node* noverlap, *noverlap_insert;
        struct node* overlap_listhead, *overlap_listcheck;
        int gindexy, gindexx, gx, gy, noverlapx, noverlapy;
        int xoverlap, yoverlap, overlap;
        char ignore_flag;

        assert(insert!=NULL);
        overlap_listhead = NULL;
#ifdef DEBUG_VB_OLG
        if(insert_flag) {     fprintf(stderr, "Inserting Overlap %c%d:\t(%5d,%5d)\n", insert->type,
insert->index, insert->x, insert->y);}
        else{                 fprintf(stderr, "Removing  Overlap %c%d:\t(%5d,%5d)\n", insert->type,
insert->index, insert->x, insert->y);}
#endif

        //get to starting row
        gindexy = insert->y/AvgRowHeight;
        assert(gindexy>=0);
        while(GridHdr[gindexy]->coordinate <= insert->y){gindexy++;}
        while(GridHdr[gindexy]->coordinate > insert->y){gindexy--;}
        insert->y = GridHdr[gindexy]->coordinate; //allign y to start of row
        gindexx = insert->x/GRID_GRAIN;
        gindexx = (gindexx>RowWidth/GRID_GRAIN)?RowWidth+1:gindexx;
#ifdef DEBUG_VB_OLGWO
        fprintf(stderr, "\tRow %d GridX %d\n", gindexy, gindexx);
#endif

        //insert, update overlap/row width
        for(gindexy; gindexy < NumRows+2 && GridHdr[gindexy]->coordinate <insert-> y + insert->height;
gindexy++)
        {
                for(gindexx; gindexx*GRID_GRAIN < insert->x + insert->width; gindexx++)
                {
                        gx = (gindexx<0)?RowWidth/GRID_GRAIN+2:gindexx;
                        gx = (gindexx>RowWidth)?RowWidth/GRID_GRAIN+1:gx;
                        gy = (gindexy<0)?NumRows+1:gindexy;
                        gy = (gindexy > NumRows)?NumRows:gy;
#ifdef DEBUG_VB_OLGWO
```

```
                                fprintf(stderr, "\tRetrivng (Y%5d,X%5d)\n", gy, gx );
#endif
                        noverlap = OverlapGrid[gy][gx];
#ifdef DEBUG_VB_OLGWO
                        if(noverlap!=NULL && noverlap->node!=NULL){ fprintf(stderr, "\tRetrieved %c
%d\n",noverlap->node->type, noverlap->node->index );}
                        else if(noverlap!=NULL) {  fprintf(stderr, "\tRetrieved NULL Node\n" ); }
                        else { fprintf(stderr, "\tRetrieved NULL\n" );}
#endif
                        noverlap_insert=NULL;

                        while(noverlap != NULL)
                        {
#ifdef DEBUG_VB_OLGWO
                                fprintf(stderr, "\tOverlap %c%d\n",noverlap->node->type, noverlap->node-
>index );
#endif

                                if(noverlap->node == insert)
                                {
                                        noverlap_insert = noverlap;
                                        noverlap = noverlap->next;
                                        continue;
                                }
                                //check to see if we already calculated overlap for noverlap
                                ignore_flag = 0;
                                overlap_listcheck = overlap_listhead;
                                while(overlap_listcheck != NULL)
                                {
                                        if(overlap_listcheck == noverlap->node) //we will ignore node and
continue through while loop
                                        {
#ifdef DEBUG_VB_OLGWO
                                                fprintf(stderr, "\t\tIgnoring Overlap %c%d\n",noverlap-
>node->type, noverlap->node->index );
#endif

                                                ignore_flag = 1; //ignored
                                                //now we must see if we can remove noverlap from the
noverlap list
                                                noverlapx = (noverlap->node->x + noverlap->node->width) /
GRID_GRAIN;
                                                 noverlapx = (noverlapx > RowWidth/GRID_GRAIN)?
RowWidth+1:noverlapx;
                                                noverlapy = (noverlap->node->y + noverlap->node-
>height)/AvgRowHeight;
                                                 while(GridHdr[noverlapy]->coordinate > noverlap->node->y)
{noverlapy--;}
                                                 while(GridHdr[noverlapy]->coordinate <= noverlap->node->y)
{noverlapy++;}
                                                if(noverlapx == gindexx && noverlapy == gindexy) //last
time we will see noverlap-> node - remove from list
                                                {
                                                        if(overlap_listcheck->north!= NULL)
{overlap_listcheck->north->south = overlap_listcheck->south;}
                                                        if(overlap_listcheck->south!=NULL)
{overlap_listcheck->south->north = overlap_listcheck->south;}
                                                                if(overlap_listcheck == overlap_listhead)
{overlap_listhead = overlap_listcheck->south;}
                                                        overlap_listcheck->north = NULL;
                                                        overlap_listcheck->south = NULL;
                                                }
                                                break; //already found noverlap - no need to move through
rest of list
                                        }
                                        overlap_listcheck = overlap_listcheck->south;
                                }
                                if(ignore_flag == 1) { noverlap = noverlap->next; continue; }

#ifdef DEBUG_VB_OLGWO
                                 fprintf(stderr, "\t\tCalculating Overlap %c%d\n",noverlap->node->type,
```

```
noverlap->node->index );
#endif

                                //we have not calculated overlap for noverlap yet - lets do it now
                                //calculate x overlap
                                //---------------------------------------- insert
                                        //-------------------- noverlap
                                if(insert->x <= noverlap->node->x &&
                                        insert->x +insert->width >= noverlap->node->x &&
                                        insert->x + insert->width >= noverlap->node->x + noverlap->node-
>width)
                                { xoverlap = noverlap->node->width;}
                                //--------------------insert
                                        //----------------------- noverlap
                                else if(insert->x <= noverlap->node->x &&
                                        insert->x +insert->width >= noverlap->node->x)
                                {xoverlap = insert->x + insert->width - noverlap->node->x; }
                                //---------------------------------------- noverlap
                                        //-------------------- insert
                                if(noverlap->node->x <= insert->x &&
                                        noverlap->node->x +noverlap->node->width >= insert->x &&
                                        noverlap->node->x + noverlap->node->width >= insert->x + insert-
>width)
                                { xoverlap = insert->width;}
                                //--------------------noverlap
                                        //----------------------- insert
                                else if(noverlap->node->x <= insert->x &&
                                        noverlap->node->x + noverlap->node->width >= insert->x)
                                {xoverlap = noverlap->node->x + noverlap->node->width - insert->x; }
                                else{xoverlap = 0;}

                                //calculate y overlap
                                //same code as above, except x->y, width->height
                                if(xoverlap==0){yoverlap = 0;}//not really, but save going through the
logic
                                 //---------------------------------------- insert
                                          //-------------------- noverlap
                                   else if(insert->y <= noverlap->node->y &&
                                           insert->y +insert->height >= noverlap->node->y &&
                                           insert->y + insert->height >= noverlap->node->y + noverlap-
>node->height)
                                   { yoverlap = noverlap->node->height;}
                                   //--------------------insert
                                           //----------------------- noverlap
                                   else if(insert->y <= noverlap->node->y &&
                                           insert->y +insert->height >= noverlap->node->y)
                                   {yoverlap = insert->y + insert->height - noverlap->node->y; }
                                   //---------------------------------------- noverlap
                                           //-------------------- insert
                                   if(noverlap->node->y <= insert->y &&
                                           noverlap->node->y +noverlap->node->height >= insert->y &&
                                           noverlap->node->y + noverlap->node->height >= insert->y +
insert->height)
                                   { yoverlap = insert->height;}
                                   //--------------------noverlap
                                           //----------------------- insert
                                   else if(noverlap->node->y <= insert->y &&
                                           noverlap->node->y + noverlap->node->height >= insert->y)
                                   {yoverlap = noverlap->node->y + noverlap->node->height - insert->y; }
                                   else{yoverlap = 0;}

                                overlap = xoverlap*yoverlap;
#ifdef DEBUG_VB_OLGWO
        fprintf(stderr, "\tFinished recalculating overlap\n");
#endif

                                //if we will encounter this node again, append it to the noverlap list to
ignore
                                noverlapx = (noverlap->node->x + noverlap->node->width) / GRID_GRAIN;
```

```
                                noverlapx = (noverlapx > RowWidth/GRID_GRAIN)?
RowWidth/GRID_GRAIN+1:noverlapx;
                                noverlapy = (noverlap->node->y + noverlap->node->height)/AvgRowHeight;
                                if(noverlapy>NumRows){noverlapy = NumRows;}
                                else
                                {
                                        while(GridHdr[noverlapy]->coordinate > noverlap->node->y+noverlap-
>node->height){noverlapy--;}
                                        while(GridHdr[noverlapy]->coordinate <= noverlap->node-
>y+noverlap->node->height)
                                        {
                                                noverlapy++;
                                                if(noverlapy>NumRows){noverlapy = NumRows; break;}
                                        }
                                }
                                if(noverlapx != gindexx || noverlapy != gindexy)
                                {
                                        noverlap->node->north = NULL;
                                        noverlap->node->south = overlap_listhead;
                                        overlap_listhead = noverlap->node;
                                }

                                if(insert_flag==0){ gridOverlap -= overlap*overlap; } //remove
                                else{ gridOverlap += overlap*overlap;} //insert

                                noverlap = noverlap->next;
#ifdef DEBUG_VB_OLGWO
        fprintf(stderr, "\tAdded to overlap list\n");
#endif


                        }//end while noverlap!=NULL

                        //clear overlap list
                        for(overlap_listcheck = overlap_listhead; overlap_listcheck!=NULL;
overlap_listcheck = overlap_listhead)
                        {
                                overlap_listhead = overlap_listcheck->south;
                                overlap_listcheck->north = NULL;
                                overlap_listcheck->south = NULL;
                        }
                        //remove/add insert to list of nodes in this grid square
                        if(insert_flag == 0) //remove
                        {
                                assert(noverlap_insert!=NULL);
                                if(noverlap_insert->next!=NULL){noverlap_insert->next->prev =
noverlap_insert->prev;}
                                if(noverlap_insert->prev!=NULL){noverlap_insert->prev->next =
noverlap_insert->next;}
                                if(OverlapGrid[gy][gx]==noverlap_insert){OverlapGrid[gy]
[gx]=noverlap_insert->next;}
                                noverlap_insert->next = NULL;
                                noverlap_insert->prev = NULL;

                        }
                        else //insert node into grid square
                        {
#ifdef DEBUG_VB_OLGWO
                                fprintf(stderr, "\tInserting Row%4d Col%4d\n", gy, gx);
#endif
                                assert(noverlap_insert==NULL);
                                do{noverlap_insert = malloc(sizeof(struct
overlap_node));}while(noverlap_insert==NULL);
                                noverlap_insert->next = OverlapGrid[gy][gx];
                                noverlap_insert->prev = NULL;
                                noverlap_insert->node = insert;
                                OverlapGrid[gy][gx] = noverlap_insert;

#ifdef DEBUG_VB_OLGWO
```

```
                                fprintf(stderr, "\tInserted Row%4d Col%4d\n", gy, gx);
#endif

                        }
                }//end column
        }//end row
        //update extra RowWidth && cleanup
        if(insert_flag==0)
        {
                if(insert->x < 0){ extraRowWidth += insert->x*insert->height; }
                else if(insert->x + insert->width > RowWidth){ extraRowWidth -= (insert->x + insert-
>width - RowWidth)*insert->height;}
                if(insert->y < 0){ extraRowWidth += insert->y*insert->width; }
                else if(insert->y + insert->width > GridHdr[NumRows]->coordinate)
                { extraRowWidth -= (insert->y + insert->height - GridHdr[NumRows]->coordinate)*insert-
>width;}
        }
        else
         {
                 if(insert->x < 0){ extraRowWidth -= insert->x*insert->height; }
                 else if(insert->x + insert->width > RowWidth){ extraRowWidth += (insert->x + insert-
>width - RowWidth)*insert->height;}
                if(insert->y < 0){ extraRowWidth -= insert->y*insert->width; }
                 else if(insert->y + insert->width > GridHdr[NumRows]->coordinate)
                 { extraRowWidth += (insert->y + insert->height - GridHdr[NumRows]->coordinate)*insert-
>width;}

         }

        return;
}
int MoveOverlapRandom(struct node* move)
{
        struct node* blocking;
        int x,y,err;
        //remove
        work_overlap(move, 0); //remove node
        //find random spot
        x = rand()%(RowWidth);
        do
        {
                y = rand()%NumRows;
        }while(GridHdr[y]->coordinate + move->height > GridHdr[NumRows]->coordinate);
        y = GridHdr[y]->coordinate;

        return InsertOverlapNode(move, x, y);
}

void AcceptOverlapMove()
{
         struct node *n, *ncpy;
        struct overlap_node *onode, *onodecpy, *onodecpyprev, *onodetmp;
         int i, j;
         int ecnt;
         //copy modules
#ifdef DEBUG_VB_OLG
        fprintf(stderr, "\tAccepted Move\n");
#endif
         for(i=0; i<Modules+1; i++)
         {
                 n = N_Arr[i];
                 ncpy = N_ArrCpy[i];
                 if(n==NULL){continue;}
                assert(ncpy != NULL);
                 CopyNode(n, ncpy);
#ifdef DEBUG_VB_OLGACC
                        fprintf(stderr, "\t\tBin Y%5dX%5d\n", i, j);
#endif
         }
```

```
#ifdef DEBUG_VB_OLGACC
        fprintf(stderr, "\t\tCopied Nodes\n");
#endif

        //copy grid ptrs
        for(i=0; i<NumRows+2; i++)
        {
                for(j=0; j<RowWidth/GRID_GRAIN+2; j++)
                {
#ifdef DEBUG_VB_OLGACC
                        fprintf(stderr, "\t\tBin Y%5dX%5d\n", i, j);
#endif

                          onode = OverlapGrid[i][j];
                        onodecpy = OverlapGridCpy[i][j];
                        onodecpyprev = NULL;
                        while(onode != NULL || onodecpy != NULL)
                        {

                                if(onode==NULL) //remove onodecpy
                                {
                                        if(onodecpy->prev!=NULL){onodecpy->prev->next = onodecpy->next;}
                                        else { OverlapGridCpy[i][j] = onodecpy->next; } //head of list
                                        if(onodecpy->next){onodecpy->next->prev = onodecpy->prev;}
                                        onodetmp = onodecpy;
                                        onodecpyprev = onodecpyprev;
                                        onodecpy = onodecpy->next;
                                        free(onodetmp);
                                        onodetmp=NULL;
                                        //onode is already NULL
                                }
                                else if(onodecpy == NULL) //add onodecpy
                                {
                                        do{onodecpy = malloc(sizeof(struct
overlap_node));}while(onodecpy==NULL);
                                        if(OverlapGridCpy[i][j]==NULL){ OverlapGridCpy[i][j] = onodecpy; }
                                        onodecpy->prev = onodecpyprev;
                                        if(onodecpyprev!=NULL){onodecpyprev->next = onodecpy;}
                                        onodecpy->next = NULL;
                                        if(onode->node->type=='a'){onodecpy->node = N_ArrCpy[onode->node-
>index];}
                                        else if(onode->node->type=='p'){onodecpy->node = N_ArrCpy[onode-
>node->index + PadOffset];}
                                        onodecpyprev = onodecpy;
                                           onodecpy = onodecpy->next; //NULL
                                        onode = onode->next;
                                }
                                else if(onode->node->type != onodecpy->node->type ||
                                                onode->node->index != onodecpy->node->index) //different
nodes
                                {
                                        if(onode->node->type=='a'){onodecpy->node = N_ArrCpy[onode->node-
>index];}
                                        else if(onode->node->type=='p'){onodecpy->node = N_ArrCpy[onode-
>node->index + PadOffset];}
                                        onodecpyprev = onodecpy;
                                        onodecpy = onodecpy->next;
                                        onode = onode->next;
                                }
                                 else
                                   {
                                           onode = onode->next;
                                        onodecpyprev = onodecpy;
                                         onodecpy = onodecpy->next;
                                   }

                        }
                }
        }
```

```
	verify();
}
void RejectOverlapMove()
{
	struct node *n, *ncpy;
	struct overlap_node *onode, *onodecpy, *onodeprev, *onodetmp;
	 int i, j;
	 int ecnt;
#ifdef DEBUG_VB_OLG
	 fprintf(stderr, "\tRejected Move\n");
#endif

	 //copy modules
	 for(i=0; i<Modules+1; i++)
	 {
		n = N_Arr[i];
		ncpy = N_ArrCpy[i];
		if(n==NULL){continue;}
		assert(ncpy != NULL);
		CopyNode(ncpy, n); //no pointer - can directly copy
	 }
#ifdef DEBUG_VB_OLGREJ
	 fprintf(stderr, "\t\tCopied Nodes\n");
#endif

	 //copy grid ptrs
	 for(i=0; i<NumRows+2; i++)
	 {
		for(j=0; j<RowWidth/GRID_GRAIN+2; j++)
		{
#ifdef DEBUG_VB_OLGREJ
		  fprintf(stderr, "\t\tBin Y%5dX%5d\n", i, j);
#endif
		  onode = OverlapGrid[i][j];
		onodecpy = OverlapGridCpy[i][j];
		onodeprev = NULL;
		while(onodecpy != NULL || onode != NULL)
		{

			if(onodecpy==NULL) //remove onode
			{
#ifdef DEBUG_VB_OLGREJ
				 fprintf(stderr, "\t\tOnode Deleted\n");
#endif

				if(onode->prev!=NULL){onode->prev->next = onode->next;}
				else { OverlapGrid[i][j] = onode->next; } //head of list
				if(onode->next){onode->next->prev = onode->prev;}
				onodetmp = onode;
				onodeprev = onodeprev;
				onode = onode->next;
				free(onodetmp);
				onodetmp=NULL;
				//onodecpy is already NULL
			}
			else if(onode == NULL) //add onode
			{
#ifdef DEBUG_VB_OLGREJ
				   fprintf(stderr, "\t\tOnodeCpy Deleted\n");
#endif

				do{onode = malloc(sizeof(struct
overlap_node));}while(onode==NULL);
				onode->prev = onodeprev;
				if(onodeprev!=NULL){onodeprev->next = onode;}
				onode->next = NULL;
				if(onodecpy->node->type=='a'){onode->node = N_Arr[onodecpy->node-
>index];}
				else if(onodecpy->node->type=='p'){onode->node = N_Arr[onodecpy-
```

```
>node->index + PadOffset];}
                                        onodeprev = onode;
                                           onode = onode->next; //NULL
                                        onodecpy = onodecpy->next;
                                }
                                else if(onodecpy->node->type != onode->node->type ||
                                                onodecpy->node->index != onode->node->index) //different
nodes
                                {
#ifdef DEBUG_VB_OLGREJ
                                           fprintf(stderr, "\t\tOnode Replaces\n");
#endif

                                        if(onodecpy->node->type=='a'){onode->node = N_Arr[onodecpy->node-
>index];}
                                        else if(onodecpy->node->type=='p'){onode->node = N_Arr[onodecpy-
>node->index + PadOffset];}
                                        onodeprev = onode;
                                        onode = onode->next;
                                        onodecpy = onodecpy->next;
                                }
                                else
                                {
                                        onodeprev = onode;
                                        onode = onode->next;
                                        onodecpy = onodecpy->next;
                                }

                        }
                }
        }
        verify();
}

int CostTimberwolf(struct node* ncost, int xorg, int yorg, int overlaporg, int row_widthorg)
{
        return Cost(ncost, xorg, yorg) +
                (GetOverlapCost()-overlaporg)*OVERLAP_WEIGHT +
                (GetExtraRowWidthCost() - row_widthorg)* ROW_WIDTH_WEIGHT;
}

void inline verify()
{
#ifdef DEBUG_VB_OLGVERIFY
        int i, j;
        struct overlap_node *onode, *onodecpy;
        for(i=0; i<NumRows+2; i++)
        {
                for(j=0; j<RowWidth/GRID_GRAIN+2; j++)
                {
                        onode = OverlapGrid[i][j];
                          onodecpy = OverlapGridCpy[i][j];
                        while(onode != NULL || onodecpy !=NULL)
                        {
                                fprintf(stderr, "\t\tVerify Bin (Y%4d,X%4d)\n", i, j);
                                assert(onode != NULL);
                                assert(onodecpy != NULL);
                                assert(onode->node->index == onodecpy->node->index);
                                onode = onode->next;
                                onodecpy = onodecpy->next;
                        }
                }
        }

#endif
}
```

Makefile

```
#https://sites.google.com/site/michaelsafyan/software-engineering/how-to-write-a-makefile
CC := gcc
program_NAME := algo
program_C_SRCS := $(wildcard *.c)
#program_CXX_SRCS := $(wildcard *.cpp)
program_C_OBJS := ${program_C_SRCS:.c=.o}
#program_CXX_OBJS := ${program_CXX_SRCS:.cpp=.o}
program_OBJS := $(program_C_OBJS) #$(program_CXX_OBJS)
program_INCLUDE_DIRS :=
program_LIBRARY_DIRS := /usr/local/lib
program_LIBRARIES := m
program_MAGICK := `pkg-config --cflags --libs MagickWand`

CPPFLAGS += $(foreach includedir,$(program_INCLUDE_DIRS),-I$(includedir))
LDFLAGS += $(foreach librarydir,$(program_LIBRARY_DIRS),-L$(librarydir))
LDFLAGS += $(foreach library,$(program_LIBRARIES),-l$(library))
LDFLAGS += $(program_MAGICK)
FLAGS :=

.PHONY: all clean distclean

all: $(program_NAME)

$(program_NAME): $(program_OBJS)
	$(CC) $(FLAGS)  $(CPPFLAGS) $(program_OBJS)  -o $(program_NAME)  $(LDFLAGS)

#algo.o: algo.c
#	$(CC) $(FLAGS) $(LDFLAGS) -c algo.c


image.o: image.c
	$(CC) $(FLAGS) -c image.c $(program_MAGICK)

clean:
	@- $(RM) $(program_NAME)
	@- $(RM) $(program_OBJS)

distclean: clean
```