Exploring Adjacency in Floorplanning*

Jia Wang
Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, IL, USA

Hai Zhou
Fudan University, China
Northwestern University, USA

***Abstract*— This paper describes a new floorplanning approach called *Constrained Adjacency Graph* (CAG) that helps exploring adjacency in floorplans. CAG extends the previous adjacency graph approaches by adding explicit adjacency constraints to the graph edges. After sufficient and necessary conditions of CAG are developed based on dissected floorplans, CAG is extended to handle general floorplans in order to improve area without changing the adjacency relations dramatically. These characteristics are currently utilized in a randomized greedy improvement heuristic for wire length optimization. The results show that better floorplans are found with much less running time for problems with 100 to 300 modules in comparison to a simulated annealing floorplanner based on sequence pairs.**

I. Introduction

Floorplanning is an important technique in VLSI circuit design since it determines the locations of the modules in the circuit on a chip. In its original meaning [1], it focuses on realizing adjacency relations for soft modules. Grason graphs [2] and rectangular dualization techniques [3], [4], [5] were developed to address the requirement for adjacency. The floorplanning flow commonly started with the structure graph [5], where the edges were the desired adjacency relations. Then, the graph was planarized and properly triangulated such that the resulting graph has a rectangular dual. Algorithms with linear complexity were developed [4] to identify the graphs that have rectangular duals and to construct the rectangular duals. Several perturbations were designed in the work [5] such that the adjacency graph could be used in iterative algorithms like simulated annealing. However, those algorithms were still complicated and not widely used today.

Then, floorplan representations emphasizing more on placement, which focuses on placing hard modules without overlap, than floorplanning [1], [6] were developed. Most of the time, simulated annealing is used to improve the floorplan according to a cost function via randomized perturbations. The drawback was that although searching for a floorplan with the least white-space was efficient by using area as the cost function, including the interconnects in the cost function may result in large time overheads in evaluating the cost function and the overheads were worsen with the increasing of the problem sizes.

So, floorplanning with adjacency relations is still preferable for connectivity centric methodologies. Returning to the rectangular dual approaches, despite the complexity involved, several issues must be addressed before their applications to current floorplanning problems. The rectangular dual consists of rectangular *room*s that contain modules. In some situations, the requirement of keeping adjacency relations may result in rooms that are significantly larger than the modules contained by them, which in turn enlarges the whitespace in the floorplan. One such case is that when a small module has several large neighbors, the room containing the small module should be large enough to realize the adjacency relations. Another issue is that although modules were thought to be soft such that they could be fit into rooms, hard modules are common in today's floorplanning problems with the introduction of hard Intellectual Property (IP) cores. We propose to allow more freedom in *NOT* keeping the adjacency relations to achieve small whitespace. The intuition behind this is that when the whitespace is small, two modules are close to each other if there are only a few modules separating them. The freedom allows us to design perturbations targeting at separating area and interconnect optimizations: first, the area optimization will not change the adjacency relations dramatically such that good relative positions for interconnects are preserved; second, the overheads of interconnect estimations are only added to the interconnect optimization.

Besides previous works on adjacency graphs [2], [3], [4], [5], adjacency relations in floorplans were captured partially in mosaic floorplan approaches and constraint graph based approaches. In mosaic floorplan approaches, e.g. Corner Block List [7] and Twin Binary Sequences (TBS) [8], as the floorplan area is divided into rectangular rooms, many adjacency relations are captured. However, since the non-crossing segment of the T-junction may slide [7], the adjacency relations among the rooms on either side of the crossing segment are not specified. On the other hand, in constraint graph based approaches Adjacent Constraint Graph (ACG) [9] and Linear Constraint Graph (LCG) [10], it was proposed to capture adjacency by removing redundancies in the constraint graph. However, being a constraint graph, there are still edges not between adjacent modules. In this work, we focus on developing a representation based on the adjacency graph, named *Constrained Adjacency Graph* (CAG). We present sufficient and necessary conditions of CAG and a linear complexity algorithm to construct *dissected floorplan*s (defined in Section II) from CAGs. After we extend CAG to handle general floorplans through packing, a "tree-weaving" algorithm and an iterative packing heuristic are developed by us to improve a CAG in area without changing the adjacency relations dramatically. The practical usages of CAG are confirmed by the experiments on floorplanning problems with 100 to 300 modules in a randomized greedy improvement heuristic, in comparison to a simulated annealing floorplanner based on sequence pairs [11].

The rest of this paper is organized as follows. In Section II, CAG is defined and the sufficient and necessary conditions are presented. Then the algorithm to construct dissected floorplans from CAGs are described in Section III. Packing based whitespace reduction techniques including the "tree-weaving" algorithm and the iterative packing heuristic are developed in Section IV. After experimental results are given in Section V, Section VI concludes the paper.

II. Constrained Adjacency Graph

A. Definitions

Given a bounding box, we can dissect it into rectangular *room*s by horizontal and vertical segments. There should be no overlap of the rooms and no empty space outside the rooms. We call this dissection a *dissected floorplan* if every room accommodates exactly one module and there is no degenerated topology where four rooms share a common point. An adjacency graph whose vertices represents the rooms can be constructed corresponding to

*The research was conducted at Northwestern University and supported in part by NSF under CNS–0613967.

the dissection: there is one edge connecting two vertices iff the two rooms are adjacent. The adjacency graph is a planar graph and when the dissection is a dissected floorplan, all the faces in the adjacency graph are triangles.

Constrained Adjacency Graph (CAG), as indicated by its name, extends the adjacency graph corresponding to a dissected floorplan by adding constraints to its edges. More formally,

Definition 1 (Constraitned Adjacency Graph): Suppose $G = (V, E)$ is a directed graph with the vertices representing rooms and the edges representing adjacencies. There are two types of edges: a vertical edge from b to t means the room t should be touched from bottom by the room b; a horizontal edge from l to r means the room r should be touched from left by the room l.

The graph G is a *Constrained Adjacency Graph* (CAG) iff there is a dissected floorplan such that there is an edge connecting two vertices iff the two rooms are adjacent and the edge describes the adjacency relationship between them.

An example of a dissected floorplan along with its CAG is shown in Fig. 1.

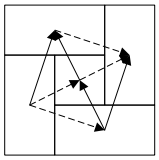

Fig. 1. A dissected floorplan with 5 rooms along with its CAG. Solid arrows are vertical adjacency relations and dashed arrows are horizontal ones.

In comparison to grason graphs [2], CAG does not need to handle the degenerated topology; comparing to rectangular dualization techniques [3], [4], [5], CAG explicitly adds constraint to the adjacency relations. These two merits make maintaining and optimizing CAGs less complicated and details will be given in the following sections.

B. Sufficient and Necessary Conditions

The necessary conditions for a graph G to be a CAG can be obtained by inspecting a dissected floorplan. Straightforwardly, a CAG should be a directed acyclic graph (DAG) and every vertex in a CAG is reachable from the vertex representing the room at the bottom-left corner. The other conditions apply to every single vertex as illustrated in Fig. 2. The detail follows.

Let E_V be the set of the vertical edges and E_H be the set of the horizontal edges. The vertical and horizontal subgraphs of G are defined as $G_V = (V, E_V)$ and $G_H = (V, E_H)$ respectively. Obviously both G_V and G_H are directed acyclic graphs (DAG). A *horizontal/vertical path* is a path whose edges are all horizontal/vertical ones. Since G is a DAG, every horizontal/vertical path is simple, i.e., it never passes one vertex more than once.

For an arbitrary vertex v, its *top edge*s are all the vertical edges leaving it and its *top neighbor*s are the vertices at the end of its top edges. From dissected floorplans, it is true that if v has at least one top neighbor, there is a unique horizontal path, denoted by $P_{top}(v)$, containing exactly all the top neighbors. Therefore, the *left-most top neighbor* and the *right-most top neighbor* can be defined as the starting vertex and the ending vertex of $P_{top}(v)$ respectively. The above definitions can be easily extend to the other three boundaries. The following lemmas hold for v in a CAG.

Lemma 1 (Neighbor Condition for v):

Suppose $P_{top}(v) = (v_1, v_2, \ldots, v_m)$. Then v_i is the bottom-most left neighbor of v_{i+1} and v_{i+1} is the bottom-most right neighbor

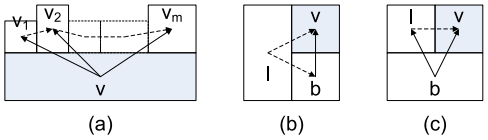

Fig. 2. (a) Neighbor condition for v; (b) (c) Corner condition for v.

of v_i for all $1 \leq i < m$. For $P_{bottom}(v)$, $P_{left}(v)$, and $P_{right}(v)$, there are similar relations.

Lemma 2 (Corner Condition for v):

The room v's bottom-left corner is not on the boundary of the dissected floorplan iff it has both bottom and left neighbors. Suppose the left-most bottom and the bottom-most left neighbor of v are b and l respectively. Then either l is the top-most left neighbor of b or b is the right-most bottom neighbor of l. There are similar relations for the other three corners.

These conditions are also sufficient for a graph to be a CAG, as stated in the following theorem.

Theorem 1: A directed graph $G = (V, E)$, whose edges E are divided into vertical edges E_V and horizontal edges E_H, is a CAG iff

1) G is a DAG and there is a vertex such that every vertex is reachable from it.
2) The neighbor condition holds for every vertex.
3) The corner condition holds for every vertex.

The proof for the necessary conditions is straightforward and so will be omitted here. The proof for the sufficient part will be given Section III by constructing dissected floorplans from the directed graphs satisfying those conditions.

III. Dissected Floorplan from CAG

When the module sizes, which are also the minimal sizes of the rooms, are given along with the CAG, we design an $O(n)$ algorithm to construct a dissected floorplan satisfying all the adjacency constraints with minimum area as presented below. Since only the conditions in Theorem 1 are used to derive the algorithm, the algorithm proves the conditions in Theorem 1 are sufficient.

For a vertex v, suppose that the width of the module to be accommodated in the room v is $v.width$ and the height is $v.height$. Define $(v.left, v.bottom)$ to be the coordinates of the bottom-left corner of the room v and $(v.right, v.top)$ to be the top-right corner. Assume $(0, 0)$ is the bottom-left corner of the floorplan and (W, H) is the top-right corner. The algorithm will determine the coordinates in the horizontal direction and the ones in the vertical direction separately. Because of the geometric symmetry, it is enough to focus on the algorithm *H-CAG-Place* that determines the coordinates in the horizontal direction only, i.e., $v.left$ and $v.right$ for all v as well as W. The algorithm is shown in Fig. 3 and the details follow.

Input: A CAG G.	
Output: The $v.left$ and $v.right$ for every v and the width W.	
1:	Add $Head_V$.
2:	$Head_V.left \leftarrow 0$.
3:	Order the vertices by their DFS discovery times of G_V from $Head_V$.
4:	For every vertex v except $Head_V$ following the order in 3:
5:	Compute $v.left$ according to the cases as shown in Fig. 5.
6:	Compute W using (4).
7:	Compute all the $v.right$s using (5).

Fig. 3. The H-CAG-Place algorithm.

First of all, a dummy vertex $Head_V$ and corresponding edges are added such that $Head_V$ is the bottom neighbor of all the vertices in G that do not have a bottom neighbor yet. The following lemma can be proved.

Lemma 3: The neighbor condition for $Head_V$ holds and every vertex in G is reachable from $Head_V$ through only vertical edges.

Now by performing a depth-first-search (DFS) [12] in G_V starting from $Head_V$ and visiting top neighbors from left to right for each vertex, the vertices can be sorted in the order of their discovery times, i.e., the times they are first discovered in the DFS. An example of a CAG with the dissected floorplan as well as the depth-first tree with the ordering are shown in Fig. 4. This order is defined as the *V-CAG order*. (Similarly the *H-CAG order* can be defined.)

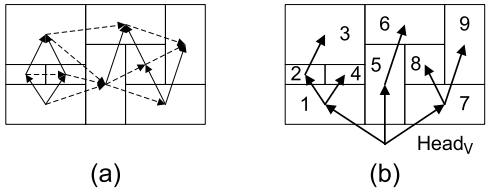

Fig. 4. (a) A CAG with the dissected floorplan; (b) The depth-first tree with vertices labeled according to the V-CAG order.

Then, the $v.left$ for each v is determined by dynamic programming following the V-CAG order. The $Head_V.left$ is set to 0 at the beginning. Since every vertex v except $Head_V$ has a bottom neighbor, there are three cases for calculating each $v.left$ as shown in Fig. 5. In the first case, the vertex v does not have a left neighbor. So the room v should be on the left boundary of the floorplan, i.e.

$$v.left = 0 \tag{1}$$

In the second case, the vertex v have both left and bottom neighbors where the bottom-most left neighbor l of v is the top-most left neighbor of the left-most bottom neighbor b of v. Now the vertex v should have the same left boundary as b, i.e.,

$$v.left = b.left \tag{2}$$

In the third case, b should be the right-most bottom neighbor of l because of the corner condition. A series of vertices $l_1, l_2, \ldots, l_k$ can be find such that $l_1 = l$, l_{i+1} is the right-most top neighbor of l_i and l_i is the right-most bottom neighbor of l_{i+1} for all $1 \leq i < k$, and either l_k does not have a top neighbor or its right-most top neighbor t does not have l_k as its right-most bottom neighbor. When l_k does not have a top neighbor, a dummy vertex t with $t.left = 0$ can be added temporarily to simplify the following procedure. Now, $v.left$ should be no less than $l_i.left + l_i.width$ for all $1 \leq i \leq k$ since they are on the two sides of a vertical segment. For the vertex b and t, $v.left$ should be no less than $b.left$ and $t.left$ to satisfy the T-junctions formed by the vertical segment and the top side of b and the bottom side of t. So, the $v.left$ in an area optimal dissected floorplan should be

$$v.left = \max\{b.left, \max_{1 \leq i \leq m}(l_i.left + l_i.width), t.left\} \tag{3}$$

The following lemmas can be proved for the validity and the complexity of the dynamic programming.

Lemma 4: In the V-CAG order, for each Equation (2) and (3), the vertices appearing at the right hand side come before the vertex appearing at the left hand side.

Lemma 5: Each vertex v appears at most $m_{top} + 2$ times (once at the left hand side, once as t, and m_{top} times as b and l_i) in Equation (1), (2), and (3) where m_{top} is the number of the top neighbors of v.

The next step is to calculate W. This is straight-forward since

$$W = \max_{v \in V}(v.left + v.width) \tag{4}$$

must be true for an area optimal floorplan.

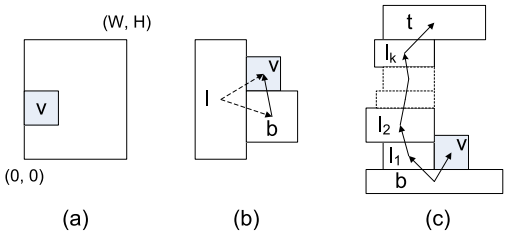

Fig. 5. Three cases for calculating $v.left$.

Finally, all the right boundaries are determined as follows.

$$\forall v \in V, v.right = \begin{cases} r.left & \text{if v has a right neighbor r} \\ W & \text{if not} \end{cases} \tag{5}$$

The algorithm called *V-CAG-Place* that determines the coordinates in the vertical direction can be derived similarly. In summary, the corresponding minimal area dissected floorplan is constructed by the H-CAG-Place algorithm and the V-CAG-Place algorithm, as stated in the following theorem.

Theorem 2: Once the CAG and the minimal room sizes are given, applying the H-CAG-Place and the V-CAG-Place algorithms constructs a dissected floorplan with the minimal area satisfying the adjacency constraints. The algorithms consume $O(n)$ time and space where n is the number of the rooms.

IV. Whitespace Reduction via Packing

Rooms could be much larger than the contained modules because of the requirements on adjacencies. In this sections, we will present techniques that reduce whitespace without change the adjacency relations dramatically.

A. Packing of Dissected Floorplans

As shown in Fig. 6, some modules in a dissected floorplan can be pushed downward since there are vertical vacancies in the rooms below them. The resulting floorplan is no longer a dissected floorplan. We call it the *V-packing* of a dissected floorplan. Similarly, the *H-packing* of a dissected floorplan is obtained by pushing all the modules leftward.

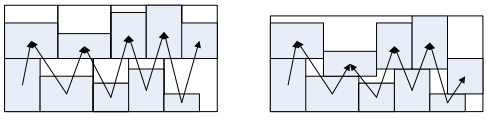

Fig. 6. (a) A dissected floorplan with the vertical subgraph G_V; (b) The V-packing of the dissected floorplan. Both rooms and modules (gray ones) are shown here.

Since G_V describes the vertical adjacency relations, the above pushing downward is formulated as the *V-CAG-Pack* algorithm as shown in Fig. 7. There is a *H-CAG-Pack* algorithm as well.

Input: A CAG G.	
Output: The $v.bottom$ and $v.top$ for every v. and the height H.	
1:	Add $Head_V$.
2:	Assign $v.height$ as the edge weight for every vertical edge (u, v).
3:	For every vertex v except $Head_V$:
4:	$v.bottom \leftarrow$ the longest path length from $Head_V$ to v in G_V.
5:	$v.top \leftarrow v.bottom + v.height$.
6:	$H \leftarrow \max_{v \in V, v \neq Head_V} v.top$.

Fig. 7. The V-CAG-Pack algorithm.

The V-CAG-Pack algorithm will not generate overlap because in a dissected floorplan, if there are two rooms whose projections to the x-axis are overlapped for a segment longer than 0, there is a vertical path from one of them to the other. This is stated as the

following lemma which includes the horizontal direction for the H-CAG-Pack algorithm as well.

Lemma 6: For two vertices u and v, if $u.left < v.left < u.right$ or $v.left < u.left < v.right$, there is a vertical path from u to v or one from v to u; if $u.bottom < v.bottom < u.top$ or $v.bottom < u.bottom < v.top$, there is a horizontal path from u to v or one from v to u.

In summary, given a CAG G, the V-packing of the dissected floorplan, written as $VP(G)$, is obtained by applying the V-CAG-Pack and the H-CAG-Place algorithm; the H-packing, written as $HP(G)$, is obtained by applying the H-CAG-Pack and the V-CAG-Place algorithm.

B. Packed Dissected Floorplans

In the V-packing of a dissected floorplan, since the modules are not packed along the horizontal direction, there are still possibilities for large whitespace. Similarly, in the H-packing of a dissected floorplan, large whitespace may appear along the vertical direction. It is not easy to perform packing on both directions simultaneously. Intuitively, if a dissected floorplan can be constructed such that it is "packed" along the horizontal direction, the whitespace after the V-packing would not be significant. The idea is formalized as follows.

Definition 2 (Packed Dissected Floorplans):

A dissected floorplan is H-packed if the H-CAG-Pack algorithm gives the same $v.left$ for every vertex v as the H-CAG-place algorithm. A dissected floorplan is V-packed if the V-CAG-Pack algorithm gives the same $v.bottom$ for every vertex v as the V-CAG-place algorithm.

In the V-CAG-Pack algorithm, since $v.bottom$ is calculated as the length of the longest path in G_V, a longest-path tree rooted at $Head_V$ can be identified, in which a vertex u is the parent of a vertex v only if (u, v) is the last edge on the longest path from $Head_V$ to v. We call this tree the ***V-LP tree*** of the CAG. Similarly the ***H-LP tree*** rooted at $Head_H$ is defined after the H-CAG-Pack algorithm.

Given a tree T rooted at $Head_H$ containing all the vertices, the *H-Tree-Weaving* algorithm as shown in Fig. 8 creates a new CAG whose H-LP tree is T and the corresponding dissected floorplan is H-packed. The details follow.

Input: A tree T rooted at $Head_H$.
Output: A CAG G^* whose H-LP tree is T.

1: $Head_H.left \leftarrow 0$;$Head_H.width \leftarrow 0$.
2: Order the vertices by their discovery times in a DFS of T from $Head_H$.
3: For every vertex v except $Head_H$ following the order in 2:
4: Find if the left-most bottom neighbor u of v exists and all the left neighbors of v according to Fig. 10.
5: If u exists:
6: Find the bottom neighbors of v starting from u as Fig. 11.
7: If (d) happens:
8: $u \leftarrow y$. Go to 6.
9: Finalize the G^*.

Fig. 8. The H-Tree-Weaving algorithm.

The new CAG G^* is created by adding vertices one by one following the order of their discovery times computed by a DFS on the tree T where the children of a vertex are visited from the bottom to the top. During the progress of the algorithm, G^* is kept as a *H-stepwise* CAG instead of a CAG in order to simplify the computation. As shown in Fig. 9, the H-stepwise CAG relaxes the constraints along the right boundary of the bounding box: instead of one vertical path from the bottom to the top along the right boundary, multiple vertical paths joined by horizontal paths are allowed, e.g., vertical paths from f to e, from d to c, and from b to a are joined by horizontal paths from d to e and from b to c. In addition, modules sizes are taken into consideration such that the rooms represented by the end point of those vertical paths can be extended to the top boundary, e.g., $d.left + d.width$ is no less than $v.left + v.width$ for every v on the path from d to c and $b.left + b.width$ is no less than $u.left + u.width$ for every u on the path from b to a. When all the vertices are added to G^*, it is *finalized* into a CAG. This finalization process can be divided into three steps. First, a dummy vertex assumed to have $Head_H$ as its parent with infinite width is added to G^* to represent the top boundary using the same method as from line 4 to line 8. After the rooms along the top boundary are identified with the help of the dummy vertex, the dummy vertex and the edges connected to it are removed. Finally, the rooms along the right boundary are extended horizontally to touch the boundary.

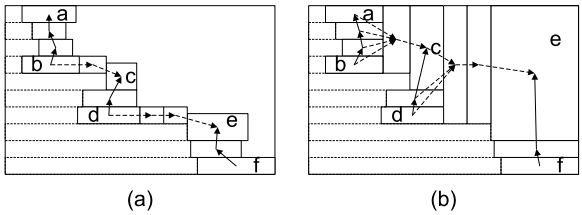

Fig. 9. (a) A H-stepwise CAG; (b) Finalize the H-Stepwise CAG into a CAG.

The details from line 4 to line 8 in order to add a vertex v to G^* are as follows. Assume $Head_H.left = Head_H.width = 0$. Suppose v's parent is p. The $v.left$ is set to $p.left + p.width$. Then, the left-most bottom neighbor u of v is determined: if v has a bottom sibling, u is that bottom sibling; if v does not has a bottom sibling, u is found by following the right-most bottom neighbor starting from p such that $u.left + u.width > v.left$; if no such u exists, v won't have bottom neighbor. The left neighbors of v are found at the same time: for the first case, only p is the left neighbor; for the two latter cases, every vertex reached except u is a left neighbor. These three cases are shown in Fig. 10.

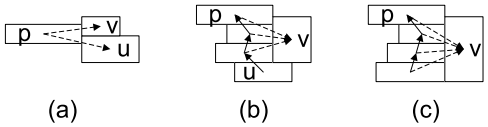

Fig. 10. Find the left-most bottom neighbor u of v: (a) v has a bottom sibling u; (b) v does not have a bottom sibling but u can be found; (c) v has no bottom neighbor. Rooms are extended accordingly for clarity.

After the left-most bottom neighbor u of v is found, all the bottom neighbor of v are found from left to right. There are four cases as shown in Fig. 11. They all starts by identifying a horizontal path from u following the top-most right neighbors. In Fig. 11 (a), the path ends at a vertex w satisfying that:

$$w.left < v.left + v.width < w.left + w.width$$

In Fig. 11 (b) and (c), the path ends at a vertex w without a right neighbor satisfying that:

$$w.left + w.width \leq v.left + v.width$$

Then a vertical path to w is identified by following the right-most bottom neighbors. For (b), every vertex x on the path does not have a right neighbor and satisfying that:

$$x.left + x.width \leq v.left + v.width$$

For (c), there is a vertex x on the path such that:

$$v.left + v.width < x.left + x.width$$

In any of the above three cases, all the vertices on the horizontal path from u to w are added as v's bottom neighbor and G^* is kept as a H-stepwise CAG. For the fourth case in Fig. 11 (d.1), there is a vertex x on the vertical path to w with a right neighbor satisfying that:

$$x.left + x.width \leq v.left + v.width$$

Suppose the top-most right neighbor of x is y. As shown in Fig. 11 (d.2), y can be extended vertically to touch v from the bottom. All the vertices on the vertical path from x to w except x should be added as the left neighbor of y on top of x. All the vertices on the horizontal path from u to w should be added as the bottom neighbor of v. Then, y is treated the same way as u and this whole process is repeated again until one of the first three cases is reached.

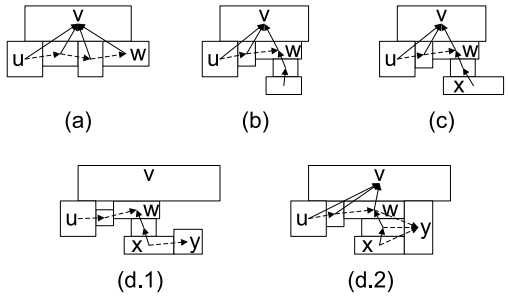

Fig. 11. Four cases for finding all the bottom neighbors of v.

Similarly, the *V-Tree-Weaving* algorithm can be derived. The following theorems state the correctness of the algorithms.

Theorem 3: The H-Tree-Weaving algorithm creates a CAG with the given H-LP tree. The V-Tree-Weaving algorithm creates a CAG with the given V-LP tree.

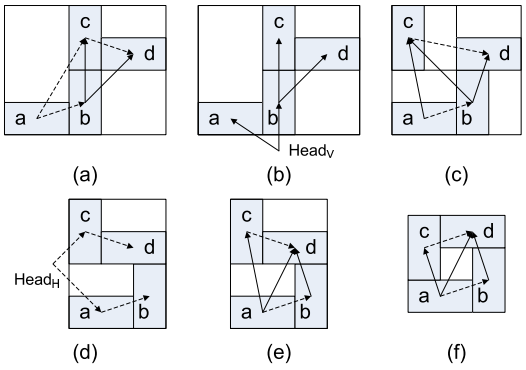

Fig. 12. Improve a dissected floorplan by applying weaving and packing alternatively. (a) The initial dissect floorplan; (b) The V-packing and the V-LP tree of (a); (c) The dissected floorplan after V-Tree-Weave; (d) The H-packing and the H-LP tree of (c); (e) The dissected floorplan after H-Tree-Weave; (f) The V-packing of (e).

C. Iterative Packing

Intuitively, when the H-LP tree is generated from a dissected floorplan whose H-packing contains little whitespace, the adjacency relations belonging to the H-LP tree are preserved and other adjacency relations will not be changed dramatically during the H-Tree-Weaving algorithms. On the other hand, the H-Tree-Weaving algorithm not only creates a H-packed dissected floorplan with a given H-LP tree: it relaxes some vertical adjacency requirements such that modules are allowed to be pushed downward further in the V-packing. This also true for the V-Tree-Weaving algorithm and an example of applying them alternatively is shown in Fig. 12.

Actually applying the two algorithm alternatively will reach an admissible placement as proposed along with the O-Tree representation [13]. In admissible placements, modules cannot be pushed either leftward or downward overlapping-freely without moving other modules. This motivates us to design the iterative packing heuristic as shown in Fig. 13 that improves the CAG in area without changing the adjacency relations dramatically. The resulting floorplan of the iterative packing heuristic is alway calculated as $HP(G)$.

Input: A CAG G.	
Output: Improved CAG G.	
1:	Compute $HP(G)$, i.e., the H-packing of the dissected floorplan.
2:	$T_0 \leftarrow$ the H-LP tree; $W_0 \leftarrow$ the width.
3:	Create a new CAG G_1 from T_0 by the H-Tree-Weaving algorithm.
4:	Compute $VP(G_1)$.
5:	$T_1 \leftarrow$ the V-LP tree; $H_1 \leftarrow$ the height.
6:	Create a new CAG G_2 from T_1 by the V-Tree-Weaving algorithm.
7:	If width of $HP(G_2) = W_0$ and height of $HP(G_2) = H_1$, return.
6:	$G \leftarrow G_2$. Go to 1.

Fig. 13. The iterative packing heuristic.

V. Experiments

A. CAG Floorplanning for Interconnects

Here we present the preliminary flow of CAG floorplanning for interconnects: an initial CAG is generated by quadratic programming and then optimized iteratively by a greedy heuristic.

Given a group of modules with the interconnects and terminals (which are the fixed pins on the floorplan boundary), the initial CAG is generated as follows to have good interconnect characteristics. First, a single quadratic programming step depending on the interconnects is used to compute the relative positions of the modules in the bounding box. The weights for the nets are all set to 1. The details of the quadratic programming can be found in placement works like [14]. After this, the modules are sorted according to their x positions and placed in a column-by-column manner from left to right: once the height of a column exceeds the square root of the total area of all the modules, a new column is created and modules are added from bottom to top again. If a dummy module representing $Head_V$ is put below all the modules, this column by column placement actually creates a V-LP tree rooted at $Head_V$ with the columns as paths in the tree. Then the V-Tree-Weaving algorithm is used to construct the initial CAG from the tree.

Once the initial CAG is obtained, we improve it iteratively through a randomized greedy improvement heuristic by applying randomized *move*s. In each iteration, the move is to randomly pick up two modules and swap them. Currently we assume that the orientations are fixed and rotations are not taken into consideration. The intuition behind these randomized moves is that the interconnects are mostly affected by the relative positions of modules when the whitespace is limited. Good relative relations can be found by following good moves using a proper cost function. Then the CAG is packed via the iterative packing heuristic to improve the area without changing the current adjacency relations dramatically. After that, the cost function is evaluated and the cost is compared to the one before this iteration. If there is any improvement, the current move is accepted and the current CAG will be the starting point of the next iteration; if there is no improvement, the current move is rejected and the CAG before this iteration is restored as the starting point. In practice, this heuristic can be terminated by adding a limit on the rejecting rate or on the running time.

B. Experimental Setup

We implemented the CAG algorithms in the C++ language. The Parquet tool [11] version 4.0 is used as a comparison. Both programs are compiled with GCC 3.4.3 and run on a Linux machine with 933MHz Pentium III processor and 512M memory.

TABLE I
STATISTICS OF THE BENCHMARKS

name	# modules	# terminals	# nets	total area
n100	100	334	885	179.5K
n200	200	564	1585	175.7K
n300	300	569	1893	273.2K

Three GSRC benchmarks with only hard modules are used: n100, n200, and n300. The statistics of these benchmarks are shown in Table I.

The Parquet tool is running in the free-outline mode with the sequence pairs representation and starting with a quadratic programming solution. The other representation in Parquet, which is B^*-tree, generates similar results according to the work [15] and thus is not compared here.

The cost function used is the weighted sum of the area and the HPWL. We use a weight ratio of $1:1$. We implement the HPWL calculation in the same manner as Parquet for fairness.

C. Experimental Results

We ran our floorplanner for 10 times with a pre-set time limit for each benchmark. For comparison, we ran Parquet twice for each benchmark: the first one is to let it run 10 times with the same time limit as ours; the second one is to let it run 10 times with a longer time limit such that the results are with the same quality as the ones in the work [15]. The results are reported in Table II. Results from our floorplanner are listed in the rows with the method "CAG" and those from Parquet are listed in the rows with the methods "Parquet A" and "Parquet B" respectively. For each benchmark and each group of the 10 runs, the "area" column shows the minimal/maximal area; the "HPWL" column shows the minimal/maximal HPWL; the "time" column shows the average running time in seconds; the "#moves" column shows the average number of the randomized moves for our approach and that of the perturbations for Parquet.

TABLE II
COMPARING CAG AND PARQUET

name	method	area	HPWL	time(s)	#moves
n100	CAG	195.9K/204.5K	302.1K/312.8K	15.30	31.3K
	Parquet A	196.8K/206.0K	320.2K/342.9K	14.90	96.5K
	Parquet B	195.0K/203.5K	313.6K/338.5K	29.80	175.8K
n200	CAG	197.0K/205.4K	540.9K/553.3K	30.86	26.0K
	Parquet A	207.4K/218.2K	613.8K/647.9K	29.40	54.0K
	Parquet B	197.4K/202.5K	578.9K/624.5K	149.2	256.6K
n300	CAG	304.4K/315.6K	649.0K/665.8K	61.61	33.8K
	Parquet A	335.2K/351.0K	750.6K/800.0K	58.89	62.5K
	Parquet B	306.9K/314.6K	709.2K/757.3K	290.6	325.7K

From the results it can be seen that the CAG approach can find better floorplans in much less time compared to the simulated annealing floorplanner Parquet. Although the CAG approach still relies on randomized moves, the far less number of moves required to reach a optimized floorplan shows that the iterative packing heuristic enables efficient explorations of the solution space when the interconnects are taken into consideration. In addition, the iterative packing heuristic does not add significant overhead when the interconnect estimation is part of the cost function. These two factors add up to the reduction in running times with better floorplans.

An optimized floorplan for n100 along with the CAG is shown in Fig. 14 to conclude this section.

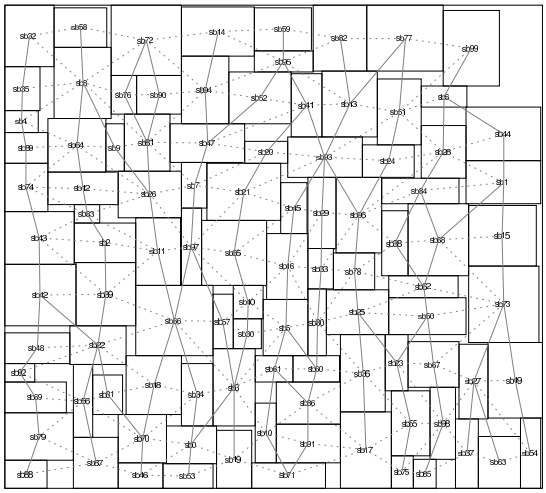

Fig. 14. A floorplan of n100 with the CAG.

VI. CONCLUSION

In this paper, we proposed to use CAG as the adjacency representation of the floorplanning problems. Algorithms were presented to construct floorplans as well as improve CAGs. A randomized greedy iterative heuristic was used to utilize the characteristics in the CAG approach and the experimental results were promising for both the quality and the running time compared to existing simulated annealing floorplanners. Research work to combine CAG in a hierarchical floorplanning framework, e.g. [16], is expected to be done in the future to fully utilize its advantages as an adjacency graph.

REFERENCES

[1] R. H.J.M. Otten. *What is Floorplan?* In ISPD, pages 201–206, 2000.

[2] J. Garson. *A Dual Linear Graph Representation for Space-Filling Location Problems of the Floor-planning Type.* G. T. Moore (ed.) Emerging Methods in Environmental Design and Planning, MIT Press, 1970.

[3] K. Kozminski and E. Kinnen. *An Algorithm for Finding a Rectangular Dual of a Planar Graph for Use in Area Planning for VLSI Integrated Circuits.* In DAC, pages 655–656, 1984.

[4] J. Bhasker and S. Sahni. *A Linear Algorithm to Find a Rectangular Dual of a Planar Triangulated Graph.* Algorithmica, 3:247–278, 1988.

[5] Y. T. Lai and S. M. Leinwand. *Algorithms for Floorplan Design via Rectangular Dualization.* IEEE Trans. on Computer-Aided Design, 7(12):1278–1289, December 1988.

[6] A. B. Kahng. *Classical Floorplanning Harmful?* In ISPD, pages 207–213, 2000.

[7] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C. Cheng, and J. Gu. *Corner Block List: An Effective and Efficient Topological Representation of Non-Slicing Floorplan.* In ICCAD, pages 8–12, 2000.

[8] F. Y. Young, C. C. Chu, and Z. C. Shen. *Twin Binary Sequences: A Non-redundant Representation for General Non-Slicing Floorplan.* IEEE Trans. on Computer-Aided Design, 22(4):457–469, April 2003.

[9] H. Zhou and J. Wang. *ACG–Adjacent Constraint Graph for General Floorplans.* In ICCD, pages 572–575, 2004.

[10] J. Wang and H. Zhou. *Linear Constraint Graph for Floorplan Optimization with Soft Blocks.* In ICCAD, pages 9–15, 2008.

[11] S. N. Adya and I. L. Markov. *Fixed-outline Floorplanning: Enabling Hierarchical Design.* IEEE Trans. On VLSI Systems, 11(6):1120–1135, December 2003.

[12] T. H. Cormen, C. E. Leiserson, R. H. Rivest, and C. Stein. *Introduction to Algorithms.* 2nd ed., MIT Press, 2001.

[13] P. N. Guo, C. K. Cheng, and T. Yoshimura. *An O-Tree Representation of Non-Slicing Floorplan and Its Applications.* In DAC, pages 268–273, 1999.

[14] N. Viswanathan and C. C. Chu. *FastPlace: Efficient Analytical Placement Using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model.* In ISPD, pages 26–33, 2004.

[15] H. H. Chan, S. N. Adya and I. L. Markov. *Are Floorplan Representations Important in Digital Design?* In ISPD, pages 129–136, 2005.

[16] J. Z. Yan and C. C. Chu. *DeFer: Deferred Decision Making Enabled Fixed-Outline Floorplanner.* In DAC, pages 161–166, 2008.