ACG–Adjacent Constraint Graph for General Floorplans*

Hai Zhou and Jia Wang
ECE, Northwestern University, Evanston, IL 60208

Abstract

ACG (Adjacent Constraint Graph) is invented as a general floorplan representation. It has advantages of both adjacency graph and constraint graph of a floorplan: edges in an ACG are between modules close to each other, thus the physical distance of two modules can be measured directly in the graph; since an ACG is a constraint graph, the floorplan area and module positions can be simply found by longest path computations. A natural combination of horizontal and vertical relations within one graph renders a beautiful data structure with full symmetry. The direct correspondence between geometrical positions of modules and ACG structures also makes it easy to incrementally change a floorplan and evaluate the result. Experimental results show the superiority of this representation.

1 Introduction

Floorplanning has seen many new representations in recent years, for example, sequence pair [11], BSG [12], O-tree [5], B*-tree [3], corner block list [6], TCG [9], twin binary sequences [15], etc. However, ACG is **not** "yet another floorplan representation". Its design results from a thorough study of different floorplan approaches (constructive or iterative) and most previous floorplan representations.

Generally speaking, there are two different approaches to floorplan design. A *constructive method* constructs a floorplan directly from module connectivity and module sizes. An *iterative method* iteratively modifies a floorplan into another and looks for a better one during the process. Using only one pass, a constructive approach is usually more efficient than its iterative counterpart. However, it is harder to design a constructive algorithm that gives comparable results as iterative approaches. Of course, iterative improvement can always be applied to results generated by a constructive approach. But a good constructive approach should make that unnecessary. With the success of early simulated annealing floorplanners, iterative approaches have become the main stream in floorplanning.

On the other hand, the research in constructive approaches is less active. Since module connectivities are used for floorplan construction, the adjacency graph (also known as Grason graph [4]) plays a crucial role in these approaches [7, 2, 8]. In an adjacency graph, an edge represents the adjacency between modules that share a boundary. Therefore, if two modules have an edge in the adjacency graph, the interconnect between them will be short. However, to be an adjacency graph, a graph must satisfy many properties: being planar, triangulated, and without complex triangles. Starting from a connectivity graph, a constructive approach will usually first create a planar graph by deleting some edges or introducing some nodes, then triangulate it and delete complex triangles. Some of these steps are NP-complete [14]. Even though there exists a linear time algorithm in theory [2], generating a floorplan from an adjacency graph is also not an easy process.

ACG bridges these two schools of floorplanning research. In iterative approaches, the generation of the floorplan from a representation usually involves a compaction (for example, from an O-tree, a module's position is found as the smallest x and y satisfying the constraints). Therefore, area is implicitly considered first. In constructive approaches, interconnects are first explored to generate an adjacency graph. They are more suitable for interconnect plan. But calculating module positions from an adjacency graph is more complex. The central concept in module compaction is a constraint graph which is used to forbid overlaps among modules. By combining adjacency graph and constraint graph, ACG represents both interconnects and areas. Since edges in an ACG only connect modules close to each other, the distance between two modules can be estimated directly in the graph. On the other hand, since an ACG is a constraint graph, the floorplan area and module positions can be efficiently evaluated by longest path computations.

As a floorplan representation, ACG is complete, non-redundant, and efficient to map to a floorplan. But we consider its main merit to be its simplicity and direct representation of geometry. This comes from a beautiful symmetrical data structure resulted from the union of horizontal and vertical edges, and based on it, a total order of modules. Because of this, incremental change in an ACG is possible and has direct meanings in the physical floorplan.

2 ACG–Adjacent Constraint Graph

The idea behind constraint graph is very simple: vertices represent modules and an edge in horizontal graph represents "left to" relation and an edge in vertical graph represents "below" relation. Adjacency graph, on the other hand, is an undirected graph and has one edge between each pair of adjacent modules. As an illustration, for a floorplan given in Figure 1(a), its constraint graph is shown in Figure 1(b) and its adjacency graph in Figure 1(c).

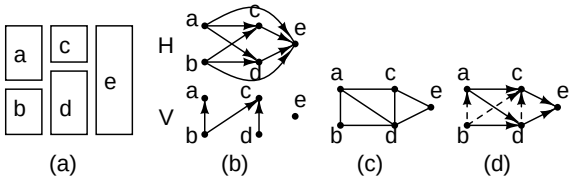

Figure 1: (a) A floorplan; (b) Constraint graph; (c) Adjacency graph; (d) ACG–Adjacent Constraint Graph.

Obviously, the adjacency graph of a given floorplan is an undirected sub-graph of its constraint graph. However, a constraint graph over-specifies. In the constraint graph in Figure 1(b), both "below" and "left to" relations are specified between modules b and c, where there is no adjacency. It is obvious that over-specification has no benefit in a representation. The essential idea of constraint graph, that is, forbidding module overlap, can be maintained by requiring that there is one relation ("left to" or "below") between any two modules.

Another set of redundant edges in a constraint graph are the transitive edges. For example, in Figure 1(b), the edge (a, e) is implied by edges (a, c) and (c, e), thus is a transitive edge. Transitive edges are not necessary for floorplan construction. On the contrary, since they are between modules far away from each other, their presence gives noises to the geometrical proximity information in other edges. For example, after deleting edges (a, e), (b, e), and one (b, c) from Figure 1(b), we arrive at a graph shown in Figure 1(d), where solid edges represent horizontal relations and dashed edges vertical ones. Notice how close this graph is to the adjacency

*This research was supported by NSF under CCR-0238484.

graph. It so happens that the graph is an ACG since there is no *cross* in it.

Definition 1 *A* cross *is a subgraph on four nodes a, b, c, d such that $(a, b), (c, d)$ are one type of edges (e.g. vertical) but $(a, c), (b, c), (a, d), (b, d)$ are the other type (e.g. horizontal).*

Definition 2 *An* ACG (Adjacent Constraint Graph) *is a constraint graph that has exact one relation (horizontal or vertical) between every pair of vertices and has no transitive edge or cross.*

We group the edges in an ACG according to the relations they represent and call them groups H and V. From the definition, there are paths between any two modules within exact one group.

ACG has many good properties which make efficient maintenance and operations possible.

Theorem 1 *The directed edges in an ACG form a total order on the vertices. In other words, the vertices can be arranged in a line such that all the edges are from left to right.*

Based on the above theorem, we organize an ACG as follows. The vertices will be doubly linked in a linear order (the total order). Edges are all directed from left to right. Each edge keeps its two end vertices and is kept in one edge list at each end vertex. Each vertex maintains four linked lists of edges: one for incoming H edges, one for outgoing H edges, one for incoming V edges, and one for outgoing V edges. The edges in each list are ordered according to the distances between end vertices: shorter edges come first. The structure is illustrated by an example in Figure 2(a), where edges are shown in arcs: the edges above the vertex line are in group H and those below are in group V. Notice that vertex d has one incoming H edge from a, one outgoing H edge to e, two incoming V edges from b and c, and no outgoing V edge. These four lists of edges have direct geometrical meanings: each connects to constraining modules in one direction: left, right, top, and bottom. And the edge orders in the lists will be either clockwise or counter-clockwise, based on how H and V edges are interpreted. For example, if the H edges are interpreted as "left to" and the V edges as "below", the geometrical interpretation of Figure 2(a) is illustrated in Figure 2(b), where H edges are ordered counter-clockwise and V edges are ordered clockwise.

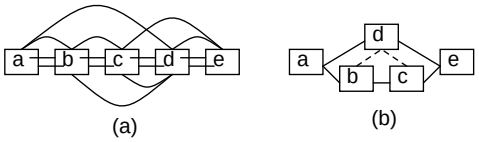

Figure 2: (a) ACG structure; (b) Geometrical relations.

One benefit of the data structure organization is that the checking and elimination of crosses are made simple and efficient. Based on Definition 1, the patterns of a cross in the data structure are shown in Figure 3. We notice here that any vertex has edges to the three other vertices. Furthermore, we can prove the following theorem.

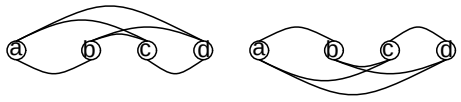

Figure 3: Patterns of a cross in an ACG.

Theorem 2 *If the cross formed on a, b, c, d is minimal, that is, no other cross exists on vertices between a and d, the vertices b, c, d are consecutive among the neighbors of a, that is, except c, no vertex between b and d is connecting to a.*

Based on this theorem, if we follow the edges of a in consecutive order, an edge type pattern "VHH" or "HVV" implies a cross. Running the checking on edges starting from every vertex can verify that there is no cross.

Corollary 2.1 *Verifying that no cross exists can be done in linear time in terms of the number of edges.*

Total symmetry is another elegant property of ACG. Formally, it can be stated as the following proposition.

Proposition 3 *Given an ACG data structure, it is still a valid ACG when the vertex order is reversed or the edge groups H and V are swapped.*

This symmetry comes directly from the symmetry in the geometrical relations represented. The four geometrical relations– left, right, up, down–are symmetrical with each other. Based on the symmetry, every provable result concerning edges in an ACG also implies the other three dual results. It simplifies our presentation in the sequel.

Corollary 3.1 *If a result concerning edges in an ACG is correct, the result by exchanging left and right (and forward and backward) or H and V is also correct.*

Given the order on vertices, the information represented by V edges is redundant to that represented by H edges. For example, in Figure 2(a), with the vertex order, one group of edges can be constructed from the other group to satisfy the ACG definition. However, keeping both groups of edges facilitates the maintenance and operations on ACG. For example, without one group of edges, it is very hard to check whether the other group belongs to a valid ACG. The following lemma shows the close relation between the two groups.

Lemma 1 *In an ACG, any two consecutive V neighbors from a given vertex are connected by an H edge.*

3 Operations on ACG

3.1 Appending

Appending is an operation to add a new vertex to the left or the right of a given ACG. We only discuss appending a vertex to the left of an ACG; appending to the right is implied by Corollary 3.1. This operation takes constant time for adding one edge and works as follows.

First, the new vertex is added to the left of the vertex linked list. Then edges from the new vertex to some other vertices are added iteratively. In each iteration, the closest vertex that does not have a relation with the new vertex is identified and a suitable type of edge is then added between them. Since the type pattern "VHH" or "HVV" gives a cross, once the edge type changes, it needs to keep changing.

The key operation in each iteration, that is, identifying the closest vertex not yet having a relation with the new vertex, can be done in constant time, thanks to the ACG structure. At the beginning, when no edge is on the new vertex, it has no relation with its right vertex. During the iterations, when the new vertex has already some edges, a relation may be implied by a path from the new vertex to another vertex. The following lemma shows that such a candidate vertex can be easily identified in all situations.

Lemma 2 *When the new vertex has only V edges, the first H neighbor of its furthest V neighbor is the closest vertex not having a relation with it. When the new vertex has last two edges in types V, H, there must be an H edge from the V*

neighbor to the H neighbor, and the closest vertex not having a relation with the new vertex is connected to the V neighbor next to that H edge.

The two situations in the lemma is illustrated in Figure 4, where the steps to find the candidate vertices are shown by the numbers.

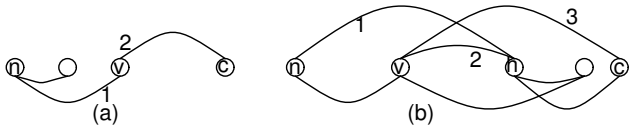

Figure 4: (a) New vertex n has only V edges; (b) New vertex n has both V and H edges.

As an example, consider to append a new vertex e to an ACG with four vertices a, b, c, d as shown in Figure 5. Vertex e is first put at the left of the vertex list. In the first iteration, d is the candidate vertex, and an edge of arbitrary type (say V) can be added from e to d. Next, since e has a V edge, the first H neighbor of d, that is c, will be the candidate. Once again, the type can be selected and suppose (e, c) is of type H. Now e has both types of edges. According to the lemma, (d, c) is an H edge, and its next edge from d is (d, b). So b is identified as the candidate. But since the edge types already alternate, edge (e, b) should be of type V–different from that of edge (e, c). The last iteration will similarly identify a and add an H edge (e, a).

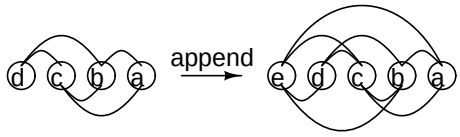

Figure 5: Append vertex e to the left of a given ACG.

Based on the discussion of the procedure, we can state the following theorem.

Theorem 4 *Given an ACG, the procedure* `append` *produces a valid ACG with one more vertex, with linear time and storage complexity in terms of added edge numbers.*

3.2 Swap

Swap is an operation that exchanges the positions of two adjacent vertices in the vertex list. Since an ACG requires edges directed from left to right, the original edge must be removed and a new edge is added to the other group. Since edges in one group represent horizontal relations and those in the other one represent vertical relations, swap will change the geometrical relation of two modules from horizontal to vertical, or vice versa. As an example, consider the ACG in Figure 2(a). If we swap vertices b, c, the result is shown in Figure 6(a). As we can see, other edges must also be modified to keep the ACG valid. The geometrical effect of the swap is shown in Figure 6(b). Module c is moved from the right of module b to below it.

Suppose a swap is done on vertices a, b and the original edge (a, b) is of type H. After the swap, edge (a, b) is deleted from group H and a new edge (b, a) is added to group V. All H paths through (a, b) are then broken, which may leave some vertex pairs without any relation. On the other hand, with the new edge (b, a) added, new V paths may formed, which may make some V edges become transitive. The swap operation will repair these damages and make the ACG a valid one again.

First, consider the transitive edges formed in V group. Based on the way the swap operation is defined, the transitive

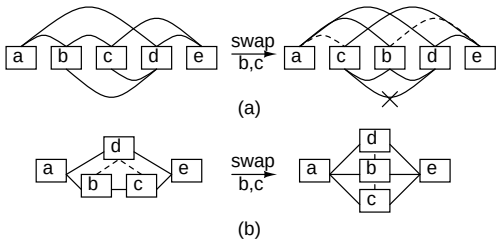

Figure 6: (a) Swap b, c; (b) The geometrical effect of the swap.

edges only appear locally. This can be stated in the following lemma.

Lemma 3 *When edge (b, a) is swapped into group V, transitive edges may only be formed from b's left V neighbors to a or from b to a's right V neighbors.*

Based on the result, we only need to check b's V left neighbors to see whether they have V edges to a. If so, these edges need to be deleted. Similarly, we will also check a's V neighbors to see whether they have V edges from b, and if so, delete them.

Then, the effects of deleting edge (a, b) from group H are considered. Two vertices will lose their relation if originally there is only one H path which goes through edge (a, b). Fortunately, the repair can also be done locally. It is easy to see that a path broken by deleting (a, b) can be restored by connecting a with the vertex after b or b with the vertex before a. Furthermore, the path is the only one between the two vertices if and only if the path between a and the vertex after b and the path between b and the vertex before a are the only paths. Therefore, we need only to consider a with b's right H neighbors and b with a's left H neighbors. However, before adding an H edge between two vertices, we must make sure that there is no other H path between them. The criteria can be stated as the following lemma. It is illustrated in Figure 7. Symmetric result exists between a and b's right H neighbors.

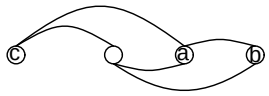

Figure 7: Vertex c has only one H path to b iff c's H neighbor before a has b as its V neighbor.

Lemma 4 *Suppose vertex c is a's left H neighbor. Then c has only one H path to b which is through (a, b) if and only if c's H neighbor before a has b as its V neighbor.*

Based on this result, for each left H neighbor of a, we will find its right H neighbor before a and check whether that vertex has b as its V neighbor. If so, we will add an H edge from the current left neighbor of a to b. Similar thing can be done with b's right H neighbors. It should be noted that a swap may introduce crosses in the graph.

4 Experiments with iterative approach

We implemented the ACG data structure and basic operations `append` and `swap` in the C language. Since `swap` may generate crosses, the implementation allows crosses in an ACG. As our focus is ACG but not the simulated annealing, we adopted an annealing scheme (starting temperature, cooling rate, etc.) similar to TCS-S [10] without further tuning. Our floorplanner works as follows. After reading in a module, it randomly `append`s the module either to the left or to the right of the

circuit	FAST-SP		TCG		TCG-S		ACG	
	area	time	area	time	area	time	area	time
apte	46.92	8	46.24	3.4	51.81	0.8	47.08	0.30
hp	11.17	9	9.02	9.9	9.56	0.6	9.33	0.24
xerox	21.07	8	19.99	3.2	20.18	0.2	20.16	0.27
ami33	1.185	28	1.20	190.5	1.21	22.6	1.20	2.15
ami49	36.82	48	37.04	590.7	38.47	63.1	36.92	6.88
n100	191.2k	128	197.8k	2487	192.0k	458	187.5k	28
n200	197.5k	350	–	>5h	197.0k	4233	187.1k	129
n300	312.3k	653	–	>5h	307.0k	16.3k	293.7k	322

Table 1: Simulated Annealing for Area (Sun Ultra 10)

circuit	FAST-SP		TCG		TCG-S	
	area	time	area	time	area	time
apte	46.92	1	46.92	1	46.92	1
hp	8.947	6	8.947	20	8.947	7
xerox	19.08	14	19.83	18	19.796	5
ami33	1.205	20	1.20	306	1.185	84
ami49	36.50	31	36.77	434	36.40	369

Table 2: Results from [13] (Sun Ultra 1) and [9, 10] (Sun Ultra 60)

current ACG, randomly deciding a relation between two modules when there is a freedom of choice. The generated ACG is used as the initial solution. Three perturbations are used in the annealing: changing the orientation of a module, exchanging two modules, and `swap`ing two adjacent modules. It can be shown that these perturbations are complete for the solution space. Each of these perturbations is selected with equal probability.

We did the experiments on a Sun Ultra 10 station over the MCNC building block benchmarks and three circuits in the GSRC floorplan benchmark suite (n100, n200, and n300). We used the default running modes targeting at area optimization for FAST-SP [13], TCG, and TCG-S, since there is no suggestion in the original papers [13, 9, 10] or in the packages. To match with the results reported in the original papers, we ran each of MCNC benchmarks 5 times for TCG and TCG-S. The results were different in different runs, since time is used for random seed set-up in TCG and TCG-S. The best result among the 5 runs and the running time for achieving that result were reported in Table 1. However, there were still some discrepancies from the results in the original papers which are copied in Table 2. For fairness, we also ran ACG in the same way (took the best result among 5 runs). Since FAST-SP does not set the random seed, it gave the same result each run. As you can see, the random seed may be one reason why we could not get the same results as in the original papers. Even on FAST-SP, which gave the same result in different runs, we also noticed that different machines gave different results. We think that it is because different systems may have different random generators. For the GSRC benchmarks, we ran each of them only once due to the large running time.

The experiments showed that ACG is faster than other representations (sequence pairs, TCG, TCG-S) while having comparable quality. The reason is simple: TCG or TCG-S has quadratic numbers of edges in the graph while ACG has only linear number of them.

We also compared ACG with Parquet-2 [1] for the three GSRC benchmarks. Each benchmark was run 5 times on the Sun Ultra 10. We ran Parquet-2 with the default parameters where the area was minimized without the restriction on the aspect ratio. And in our annealing scheme of ACG, the number of the perturbations tried in each iteration was cut by half for faster speed at the cost of solution quality. The best result among the 5 runs and the running time for achieving it were shown in Table 3. The total numbers of moves (#move) in the annealings are also reported. It showed that ACG is faster than Parquet-2 while giving better solutions.

circuit	Parquet-2			ACG		
	area	#move	time	area	#move	time
n100	194.9k	112k	17.15	187.7k	139k	13.23
n200	194.2k	224k	84.28	188.7k	278k	61.85
n300	304.1k	337k	217.6	296.3k	426k	148.7

Table 3: ACG vs. Parquet-2 (Sun Ultra 10)

References

[1] S.N. Adya and I.L. Markov. Fixed-outline floorplanning : Enabling hierarchical design. *IEEE Trans. on VLSI Systems*, 11(6):1120–1135, December 2003.

[2] J. Bhasker and S. Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. In *DAC*, pages 108–114, 1986.

[3] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu. B*-trees: A new representation for non-slicing floorplans. In *DAC*, pages 458–463, 2000.

[4] J. Grason. *A Dual Linear Graph Representation for Space-Filling Location Problems of the Floor-planning Type.* MIT Press, Cambridge, MA, 1970.

[5] 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.

[6] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu. Corner block list: An effective and efficient topological representation of non-slicing floorplan. In *ICCAD*, pages 8–12, 2000.

[7] K. Kozminski and E. Kinnen. Rectangular dual of planar graphs. *Networks*, 15:145–157, 1985.

[8] Y. T. Lai and S. M. Leinwand. Algorithms for floor-plan design via rectangular dualization. *IEEE TCAD*, 7(12):1278–1289, 1988.

[9] J.-M. Lin and Y.-W. Chang. TCG: A transitive closure graph-based representation for non-slicing floorplans. In *DAC*, pages 764–769, 2001.

[10] J.-M. Lin and Y.-W. Chang. Tcg-s: Orthogonal coupling of p*-admissible representation for general floorplans. In *DAC*, pages 842–847, New Orleans, LA, June 2002.

[11] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing based module placement. In *ICCAD*, pages 472–479, 1995.

[12] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module placement on BSG-structure and IC layout applications. In *ICCAD*, pages 484–493, 1996.

[13] X. Tang and D. F. Wong. FAST-SP: A fast algorithm for block placement based on sequence pair. In *ASP-DAC*, pages 521–526, 2001.

[14] G. Yeap and M. Sarrafzadeh. Floorplanning by graph dualization: 2-concave rectilinear modules. Manuscripts, 1993.

[15] F. Y. Young, C. C. N. Chu, and Z. C. Shen. Twin binary sequences: A non-redundant representation for general non-slicing floorplan. *IEEE TCAD*, 22(4):457–469, April 2003.