A Fast Incremental Cycle Ratio Algorithm[*]

Gang Wu and Chris Chu
Department of Electrical and Computer Engineering, Iowa State University, IA
{gangwu, cnchu}@iastate.edu

ABSTRACT

In this paper, we propose an algorithm to quickly find the maximum cycle ratio (MCR) on an incrementally changing directed cyclic graph. Compared with traditional MCR algorithms which have to recalculate everything from scratch at each incremental change, our algorithm efficiently finds the MCR by just leveraging the previous MCR and the corresponding largest cycle before the change. In particular, the previous MCR allows us to safely break the graph at the changed node. Then, we can detect the changing direction of the MCR by solving a single source longest path problem on a graph without positive cycle. A distance bucket approach is proposed to speed up the process of finding the longest paths. Our algorithm continues to search upward or downward based on whether the MCR is detected as increased or decreased. The downward search is quickly performed by a modified Karp-Orlin algorithm reusing the longest paths found during the cycle detection. In addition, a cost shifting idea is proposed to avoid calculating MCR on certain type of incremental changes. We evaluated our algorithm on both random graphs and circuit benchmarks. A timing-driven detailed placement approach which applies our algorithm is also proposed. Compared with Howard's and Karp-Orlin MCR algorithm, our algorithm shows much more efficiency on finding the MCR in both random graphs and circuit benchmarks.

1. INTRODUCTION

Given a directed cyclic graph and each edge in the graph is associated with two numbers: *cost* and *transition time*. Let the cost (respectively, transition time) of a cycle in the graph be the sum of the costs (respectively, transition times) of all the edges within this cycle. Assuming the transition time of a cycle is non-zero, the ratio of this cycle is defined as its cost divided by its transition time. The maximum cycle ratio (MCR) problem finds the cycle whose ratio is the maximum in a given graph [1]. The MCR problem is closely related to the optimization of VLSI circuits. In particular, for synchronous circuits, MCR reflects the optimization potential of the retiming or clock skew scheduling techniques being applied to the circuits [2]. For asynchronous circuits, MCR directly corresponds to the circuit performance [3]. There are also applications in other areas, e.g., time separation analysis of concurrent systems, graph theory [4].

[*]This work is supported in part by NSF award CCF-1219100.

ISPD '17, March 19-22, 2017, Portland, OR, USA
© 2017 ACM. ISBN 978-1-4503-4696-2/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3036669.3036670

In practice, most of the optimization processes which apply MCR algorithms are actually performed incrementally. For example, during the detailed placement stage of VLSI circuits, one step of the algorithm adjusts the coordinates of only a few cells. Then, evaluation is performed for this modified circuit before the next move [5] [6]. Similarly, in the gate sizing process of circuits, the algorithm might adjust the size of one gate at a time, instead of changing the sizes of all the gates at once [7]. Considering the above type of applications which only few changes are made at each step, the MCR algorithm might also be able to do the calculation "incrementally" by leveraging the information calculated at the previous step, and therefore be able to find the MCR much faster. In this paper, we focus on the MCR problem considering such incremental changes, which we referred to as the incremental MCR problem. By leveraging the previously calculated information, we expect the incremental MCR algorithm to be faster than traditional MCR algorithms, which have to recalculate everything from scratch at each step.

The MCR problem without considering the incremental changes has been well studied [1] [4] [8]. One way to solve the MCR problem is by linear programming [9]. In addition, various MCR algorithms are proposed to solve the problem more efficiently. Experimental study of existing MCR algorithms shows the Karp and Orlin's algorithm (KO) [10] and an efficient implementation of KO [11] is the fastest among them [4]. When the graph size is small, the Howard's algorithm (HOW) [12] is also able to generate comparable results [1]. For the incremental MCR problem, only very few researches have been done. In [13], the authors developed an adaptive negative cycle detection algorithm and incorporated it into the Lawler's MCR algorithm [14]. However, the experiments in [13] are performed only on very small graphs, and thus the efficiency of the algorithm cannot be confirmed. In addition, Lawler's algorithm finds MCR based on the binary search idea, which is much slower compared with KO and HOW [4].

In this paper, we propose an efficient incremental MCR algorithm. The only information we need to leverage is the previous MCR and the corresponding largest cycle in the graph before the incremental change is made. Our algorithm contains three parts: cycle detection, local upward search and global downward search. After an incremental change is made on the given graph, the cycle detection is performed first. During the cycle detection, we use our cost shifting idea to filter out the cases which the incremental change will not affect the MCR. For the remaining cases which affect the MCR, the algorithm continues to detect whether the MCR is increased or decreased. If the MCR is detected to be increased, we perform the local upward search to identify the new MCR in the changed graph. Otherwise, we perform the global downward search to identify the new MCR. To speed up the cycle detection and the local upward search, we propose a bucket distance idea which can quickly build a longest path tree in a graph without positive cycle. Also, we propose a modified KO algorithm to speed up the global downward search by reusing the longest paths found in the cycle detection step. We evaluate our algorithm on both random graphs and the ISPD 2005 placement benchmarks [15]. To evaluate our algorithm on circuit benchmarks, we propose a timing-driven detailed placement approach which applies our incremental MCR algorithm. The experimental results show our algorithm is very efficient on calculating MCR compared with the fastest traditional MCR algorithms.

The rest of this paper is organized as follows. In Section II, we briefly review the Howard's algorithm and the Karp-Orlin algorithm. In Section III, we present our incremental MCR algorithm. In Section IV, we present our timing-driven detailed placement approach. Finally, the experimental results are shown in Section V.

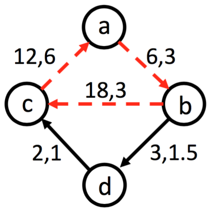
2. PRELIMINARIES

2.1 Maximum cycle ratio problem

We formally define the MCR problem in this section. Let $G = (V, E)$ be a directed cyclic graph. Each edge $e \in E$ is associated with a cost denoted as $w(e)$ and a transition time denoted as $t(e)$. Let c denotes a cycle in G. Let $\tau(c)$ denotes the cycle ratio of c. With the assumption that $\sum_{\forall e \in c} t(e) > 0$, the MCR problem finds the maximum $\tau(c)$ $\forall c \in G$ as:

$$\tau^*(G) = \max_{c \subset G} \left\{ \frac{\sum_{\forall e \in c} w(e)}{\sum_{\forall e \in c} t(e)} \right\}$$

Here, we use $\tau^*(G)$ to denote the MCR of G. As an example, Fig. 1 shows a graph with two cycles (a, b, c) and (a, b, d, c). The two numbers associated with each edge denote its cost and transition time respectively. By calculating the ratio of both cycles, we can identify the largest cycle shown as the dotted lines in Fig. 1. However, for larger graphs, it is difficult for us to enumerate all the cycles to find out which one is the largest, as the total number of cycles can be exponential to the graph size.



$$\tau(a,b,c) = \frac{6+18+12}{3+3+6} = 3$$

$$\tau(a,b,d,c) = \frac{6+3+2+12}{3+1.5+1+6} = 2$$

Figure 1: Finding the maximum cycle ratio in a graph.

One way to solve the maximum cycle ratio problem is to formulate it as a linear program [9]:

$$\begin{aligned} &\text{Minimize} \quad \tau \\ &\text{Subject to} \quad d(i) + w(i,j) - t(i,j) * \tau \le d(j) \quad \forall (i,j) \in E \end{aligned}$$

Here, (i, j) denotes the edge connecting node i to node j. $d(i)$ and $d(j)$ are free variables for each node $v \in V$ denoting its *distance.*

In addition to the linear program solution, various algorithms have been proposed and are able to solve the problem more efficiently. We discuss these algorithms below.

2.2 Traditional maximum cycle ratio algorithms

Given a cycle ratio τ, we can construct another graph $G_\tau = (V, E_\tau)$ based on G. G_τ is identical to G, except now each edge in E_τ is associated with only one number *length*, instead of having two numbers (i.e. cost and transition time). Each $e \in E_\tau$ will have a corresponding edge $e' \in E$, and the *length* of e is defined as $l(e) = w(e') - \tau * t(e')$. Correspondingly, the *length* of a cycle $c \in G_\tau$ is defined as $l(c) = \sum_{\forall e \in c} l(e)$.

G_τ has many interesting features which can help us identify the largest cycle in G. In particular, if G_τ contains positive length cycles, it means the given cycle ratio τ is less than $\tau^*(G)$. If G_τ contains zero length cycles and does not contain positive length cycles, the given τ will be equal to $\tau^*(G)$, and the zero length cycle in G_τ will correspond to the most critical cycle in G. If G_τ only contains negative cycles, it means the given cycle ratio τ is larger than $\tau^*(G)$. In this case, single source longest path trees rooted at any node $v \in V$ exist in G_τ. Detailed proofs of these facts can be found in [4].

Most of the MCR algorithms use the above facts to transfer the MCR problem into the problem of either detecting positive cycles in G_τ or maintaining a longest path tree in G_τ. Howard's algorithm and Karp-Orlin algorithm are two of the fastest algorithms among them, and they tackle the MCR problem in exactly opposite directions. In particular, Howard's algorithm starts with a very small τ and gradually increases τ until it cannot detect a positive length cycle in G_τ. Karp and Orlin's algorithm starts with a very large τ and gradually decreases τ while maintaining a longest path tree in G_τ.

2.2.1 *Howard's algorithm (HOW)*

HOW can be separated into two phases: the discovery phase and the verification phase. If the starting cycle ratio τ is small enough, all the cycles in G_τ will have a positive length. In the discovery phase, an arbitrary positive length cycle $c \in G_\tau$ is located, and we increase τ to τ' such that $l(c) = 0$ in $G_{\tau'}$. Next, in the verification phase, a positive cycle detection algorithm (e.g., the Bellman–Ford algorithm) can be used to check if there are still positive cycles in $G_{\tau'}$. If so, we repeat the discovery phase. If not, we are safe to exit the algorithm and output τ' as $\tau^*(G)$. More details and pseudo code for HOW can be found in [1] [4].

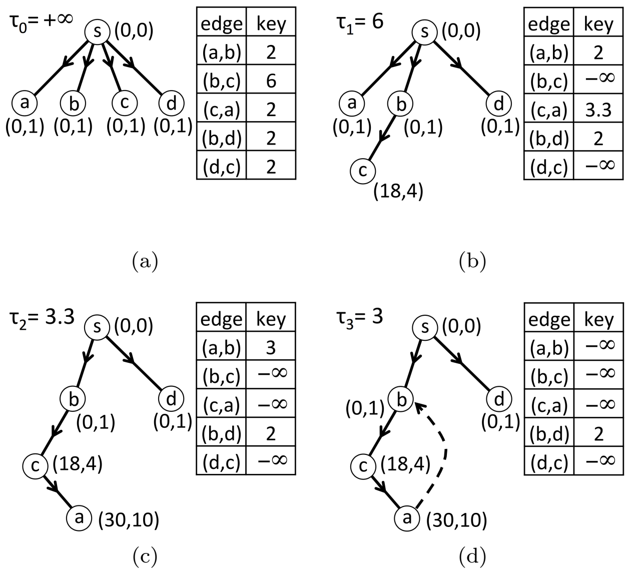
2.2.2 *Karp and Orlin's algorithm (KO)*

KO starts with a large enough τ such that all cycles in G_τ is negative and thus longest paths are well defined in G_τ. Here, we use a simple example to illustrate the basic idea of KO. For more details, please refer to [1] [4] [11].

In the beginning, KO modifies G by adding a node s and a set of edges E_s connecting s to all nodes $v \in V$, with $w(e) = 0$ and $t(e) = 1$ for all $e \in E_s$. Let a path from s to node v in G be denoted by $p(s, v)$, which corresponds to the path from root s to v in the longest path tree T_s in G_τ. For each node $v \in V$, we have $w(v) = \sum_{\forall e \in p(s,v)} w(e)$ and $t(v) = \sum_{\forall e \in p(s,v)} t(e)$, shown as $(w(v), t(v))$ in Fig. 2. For each edge $(i, j) \in E$, let $\Delta w(i, j) = w(i) + w(i, j) - w(j)$ and $\Delta t(i, j) = t(i) + t(i, j) - t(j)$. Then, a max heap containing all the edges in G is maintained using the key value calculated as follows:

$$key(i,j) = \begin{cases} \Delta w(i,j)/\Delta t(i,j), & \text{if } \Delta t(i,j) > 0. \\ -\infty, & \text{otherwise.} \end{cases}$$

Fig. 2 shows the process of calculating $\tau^*(G)$ using KO for the graph G shown in Fig. 1. Fig. 2(a) shows the initial longest path tree T_s in G_{τ_0} and the corresponding max heap. Next, edge (b,c) which has the maximum key value is retrieved from the heap, and T_s is updated by replacing edge (s,c) with (b,c) as shown in Fig. 2(b). This tree update makes T_s to be the longest path tree in G_{τ_1}. We will continue the max heap update and tree update until a cycle is found which gives us $\tau^*(G)$, as shown in Fig. 2(d).



edge	key
(a,b)	2
(b,c)	6
(c,a)	2
(b,d)	2
(d,c)	2

edge	key
(a,b)	2
(b,c)	$-\infty$
(c,a)	3.3
(b,d)	2
(d,c)	$-\infty$

edge	key
(a,b)	3
(b,c)	$-\infty$
(c,a)	$-\infty$
(b,d)	2
(d,c)	$-\infty$

edge	key
(a,b)	$-\infty$
(b,c)	$-\infty$
(c,a)	$-\infty$
(b,d)	2
(d,c)	$-\infty$

Figure 2: An example of Karp and Orlin's algorithm

3. PROPOSED ALGORITHM

We define an incremental change on an edge as a cost change on this edge, and an incremental change on a node as the cost changes on all input and output edges directly connected to the node. The transition times remain to be the same for both the edge change and the node change. In Section III-A, we first look into details about how MCR is affected by an edge change. In Section III-B, we consider the incremental changes happened on a node, which is the assumption made by our algorithm.

3.1 Considering incremental changes on an edge

Let e denotes the changed edge. We use C_e to denote the set of cycles passing through e, and when $w(e)$ changes, only the cycles in C_e will be affected. In addition, we use G to denote the graph before the change and G' to denote the graph after the change, with corresponding largest cycle to be c^* and $c^{*\prime}$ respectively. Based on whether $w(e)$ is decreased or increased and whether $e \in c^*$ or not, we can separate the incremental changes into the following four cases:

3.1.1 $e \notin c^*$ and $w(e)$ is decreased

This is the easiest case, as it can be guaranteed that $c^{*\prime} = c^*$ after the incremental change. Before the change happens, we know $\tau(c) \leq \tau(c^*)$ $\forall c \in C_e$. Since decreasing the cost of e will only decrease $\tau(c)$ $\forall c \in C_e$, none of these cycles will get a chance to become larger than c^*. Thus, c^* will remain to be the largest cycle in G'.

3.1.2 $e \notin c^*$ and $w(e)$ is increased

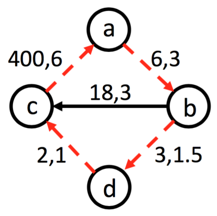
Increasing $w(e)$ will increase $\tau(c)$ $\forall c \in C_e$. It is possible that $\tau(c)$ of a cycle $c \in C_e$ becomes larger than $\tau(c^*)$ and thus dominates all other cycles and becomes the largest cycle in G'. If this happens, we have $c^{*\prime} = c$, and thus it can be guaranteed that $c^{*\prime}$ is passing through e.

3.1.3 $e \in c^*$ and $w(e)$ is decreased

Decreasing $w(e)$ will decrease $\tau(c^*)$. Thus, another cycle in G can replace c^* and becomes dominating in G'. If this happens, there is no clue for us to know where this new largest cycle is located.

3.1.4 $e \in c^*$ and $w(e)$ is increased

Increasing $w(e)$ will increase $\tau(c^*)$ and also increase $\tau(c)$ $\forall c \in C_e$. Thus, it is guaranteed that $c^{*\prime}$ is passing through e. However, there is no guaranteed that $c^{*\prime} = c^*$. As an example, let the graph in Fig. 1 to be G and the graph in Fig. 3 to be G'. It can be seen that after increase $w(c,a)$ from 12 to 400, the largest cycle also get changed.



$$\tau(a,b,c) = \frac{6+18+400}{3+3+6} = 35.3$$

$$\tau(a,b,d,c) = \frac{6+3+2+400}{3+1.5+1+6} = 35.7$$

Figure 3: $e \in c^*$ and $w(e)$ is increased.

3.2 Considering incremental changes on a node

Only considering changes on a single edge is certainly not enough, as applications typically involve multiple edge changes. We can transform the multiple edge changes into single edge change by only processing one edge at a time, but this can slow down the incremental MCR algorithm. Therefore, instead of only considering a single edge change, our algorithm handles the case of a single node change, which makes it more suitable for real applications. A single node change can create more complicated situations compared with an edge change, as cycle ratio of some cycles can decrease while others can increase at the same time. However, similar to the edge change, only the cycles passing through the changed node will be affected. If the change is happened on more than one node, our algorithm will just transform it to the single node change by processing one node change at a time.

3.3 Considering HOW and KO incrementally

HOW and KO are not suitable to perform the MCR calculation incrementally. One reason is that most of the middle information (i.e. node distances, the longest path tree) is calculated based on G_τ whose edge lengths depend on the parameter τ. Once τ is changed, all edge lengths in G_τ will be updated and the middle information calculated in the previous iteration will become useless. It is also not realistic to keep these middle information for each possible τ value, as the possible τ values correspond to all cycles in the graph whose total number is exponential to the graph size. Another reason is that, the cycle ratio can change in both directions when an incremental change is made, while HOW or KO can only search from one direction. In particular, if MCR is decreased, it will be difficult for HOW to go backward and locate the new largest cycle. Similarly for KO when MCR is increased. This suggests the incremental MCR algorithm needs to be able to search from both directions. When MCR is increased, the algorithm can search upward starting from the previous MCR, similar to HOW. When MCR is decreased, the algorithm can search downward similar to KO. This is just the basic idea of our algorithm, which we will discuss below.

3.4 An overview of our proposed algorithm

An overview of our incremental MCR algorithm is shown in Fig. 4. Give an initial graph G with its MCR to be $\tau^*(G)$ and the corresponding largest cycle to be c^*. Assuming a node v in G is updated and the set of cycles passing through v is C_v, our algorithm first detects the changing direction of MCR. If the MCR is detected as increased, we perform a local upward search to identify the new MCR. If the MCR is detected as decreased, we perform a global downward search to identify the new MCR. The output of our algorithm is $\tau^*(G')$ and $c^{*\prime}$ for G' which denotes the graph after the change.

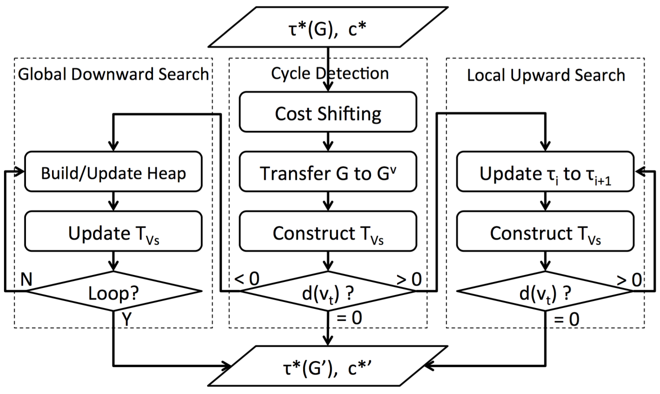


Figure 4: Overview of our incremental MCR algorithm.

3.5 Cycle detection

In the beginning, at our cost shifting step, we filter out the incremental change which will not affect MCR. If this is the case, we can directly exit the MCR algorithm and output $\tau^*(G)$ as $\tau^*(G')$. If the change has a potential to affect MCR, we continue to detect whether the MCR is increased or decreased. To do this, we first transform G into G^v by replacing node $v \in V$ with two new node v_s, v_t. Next, we build the longest path tree T_{v_s} rooted at v_s. Finally, based on the longest distance from v_s to v_t, we will be able to detect the changing direction of MCR.

3.5.1 Cost shifting

As we discussed in Section III-A case 1), if the changed edge is not on c^* and the edge cost is decreased, we can guarantee that the MCR will not be affected. The idea of cost shifting is to transform all edge changes into this particular case, by shifting edge costs from the input (or output) edges of v to the output (or input) edges of v. As an example, Fig. 5(a) shows the current edge cost changes of v with 4 decreased edges and 1 increased edge. By shifting 9 units of cost from the output edges of v to the input edges of v, we get the new cost changes shown in Fig. 5(b). Assuming $v \notin c^*$, since only decreased edges exist after cost shifting, this change of v can be identified as not affecting MCR.

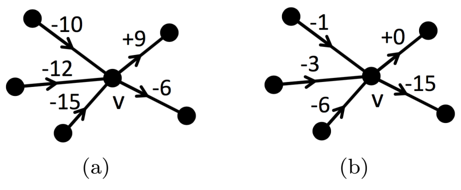


Figure 5: (a) Before the cost shifting. (b) After the cost shifting.

In general, by applying the cost shifting idea, we can exit the MCR algorithm if $v \notin c^*$ and the increment change belongs to one of the following two cases: (1) all edge costs are decreased. (2) all edge costs on one side (input or output) of v are decreased, and the smallest amount of decreasing at the decreased side is larger than largest amount of increasing on the other side.

3.5.2 Transform G into G^v

If the incremental change has a potential to affect MCR, we continue to this step and transform G into G^v as follows. We remove v from G and add two new nodes v_s, v_t to G, with v_s connecting to all v's output edges and v_t connecting to all v's input edges as shown in Fig. 6.

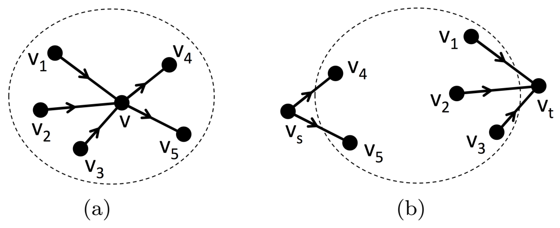


Figure 6: (a) Before breaking at v. (b) After breaking at v.

Let $\tau_0 = \tau^*(G)$, we can get the corresponding $G^v_{\tau_0}$ for G^v. Let T_{v_s} denotes the longest path tree rooted at v_s. Then we can have the following Theorem:

Theorem 1. T_{v_s} is well defined in $G^v_{\tau_0}$.

Proof: Before the incremental change, we have $l(c) \leq 0 \ \forall c \in G_{\tau_0}$. After the incremental change, only the cycles passing through v can be positive in G_{τ_0}. Since all $c \in C_v$ is broken at v in $G^v_{\tau_0}$, they will not form a positive cycle in $G^v_{\tau_0}$. Therefore, we have $l(c) \leq 0 \ \forall c \in G^v_{\tau_0}$, and thus the longest paths in $G^v_{\tau_0}$ are well defined. □

3.5.3 *Constructing T_{v_s} on $G^v_{\tau_0}$*

Constructing T_{v_s} on $G^v_{\tau_0}$ is equivalent to the problem of finding a single source longest path tree in a graph without positive cycle. Since $G^v_{\tau_0}$ contains both negative and positive length edges, Dijkstra's algorithm is not applicable here. One way to construct T_{v_s} is to use the Bellman–Ford algorithm and update the node distances in a breath first search manner, as suggested in [16]. However, the breath first search has a very limited control on the updating order of the nodes, and thus each node can be repeatedly updated for many times [17]. Therefore, the runtime of this approach is not good.

We propose a distance bucket approach to help us update the nodes in an appropriate order, which can effectively reduce the total number of updates on each node and therefore speed up the process of constructing T_{v_s}. The basic idea of the distance bucket approach is similar to the Dijkstra's algorithm: we always pick the node which has the largest distance to update. Different from Dijkstra's algorithm, this cannot guarantee that the updated node will not be updated again, but the chance that this node get updated again will be much smaller compared with a random updating order. Instead of maintaining a priority queue to exactly find the node with largest distance, we only differentiate the nodes by putting them into certain buckets based on the range of their distance. One reason we do this is that it is not necessary to differentiate the node distances exactly, as repeated update of the nodes cannot be avoided anyway. Another reason is that maintaining a priority queue is expensive, especially considering the total number of edge update operations is huge.

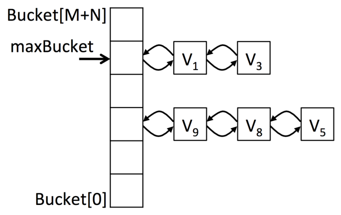


Figure 7: The distance bucket data structure.

Assuming we have $M+N$ buckets denoted from *bucket*[0] to *bucket*$[M+N-1]$ with M buckets for negative distances and N buckets for positive distances, as shown in Fig. 7. Let d_u be a unit range of distance covered by a bucket. Then, for a particular node v, its corresponding bucket index can be calculated as $M + d(v)/d_u$. Instead of storing a copy of all the contained nodes, the distance bucket only pointing to one of the contained node as shown in Fig. 7. The rest of the contained nodes will simply be connected to the this node in a doubly linked list manner. The details of our distance bucket approach is shown in Algorithm 1.

3.5.4 *Detecting the changing direction of MCR*

After constructing T_{v_s}, we can get $d(v_t)$ which is the longest distance from v_s to v_t in $G^v_{\tau_0}$. If $d(v_t) > 0$, it means a positive cycle passing through v exists in G_{τ_0}, and $\tau^*(G) < \tau^*(G')$. Therefore, we search upwards to find the new MCR. If $d(v_t) = 0$, it means $\tau^*(G) = \tau^*(G')$ and c^* remains to be the largest cycle in G'. So we can exit the MCR algorithm. If $d(v_t) < 0$, it means the largest cycle passing through v in G_{τ_0} is negative. If $v \notin c^*$, we can exit the MCR algorithm as the MCR will not be affected in this case. Otherwise, it means $\tau^*(G) > \tau^*(G')$ and we search downwards to find the new MCR.

Algorithm 1 The distance bucket approach

```
Ensure: Constructing $T_{v_s}$ on $G^v_{\tau_0}$.
 1: Insert $s$ to $bucket[0]$ and set $max := 0$;
 2: while $max > 0$ do
 3:     Pick and delete node $u$ from the $bucket[max]$.
 4:     for each $(u, v) \in E_{\tau_0}$ do
 5:         if $d(v) < d(u) + l(u, v)$ then
 6:             $d(v) := d(u) + l(u, v)$;
 7:             Find the bucket index $i$ based on $d(v)$;
 8:             if $v$ is not in any buckets then
 9:                 Insert $v$ to $bucket[i]$;
10:             else
11:                 Delete $v$ from its current bucket;
12:                 Insert $v$ to $bucket[i]$;
13:             end if
14:             if $i > max$ then
15:                 $max := i$;
16:             end if
17:         end if
18:     end for
19:     while $bucket[max]$ is empty do
20:         $max := max - 1$
21:     end while
22: end while
```

3.6 Local upward search

In this step, we search upwards until $\tau^*(G')$ is identified. It is safe for us to only perform a local search among all the cycles in C_v based on the following theorem:

Theorem 2. If $\tau^*(G') > \tau^*(G)$, $c^{*\prime} \in C_v$.

Proof: In Section III-A, the only cases which the incremental change can increase the MCR are case 2) and case 4). As we have discussed, we can guarantee $c^{*\prime}$ is passing through the changed edge e in both these two cases. Since e is connected to v, this means $c^{*\prime}$ must also pass through v. □

The strategy we used to perform the local upward search is similar to HOW. Assuming the current cycle ratio is τ_i, we first increase τ_i to τ_{i+1}, which makes $d(v_t) = 0$ in current T_{v_s}. Next, we construct the new T_{v_s} in $G^v_{\tau_{i+1}}$ using Algorithm I and get the corresponding new $d(v_t)$. If $d(v_t) > 0$, it means there are still positive cycles existing in $G_{\tau_{i+1}}$ whose cycle ratio is larger than τ_{i+1}. So we repeat the first step and keep updating the cycle ratio. Otherwise, we can exit the MCR algorithm and output τ_{i+1} as $\tau^*(G')$.

3.7 Global downward search

Our algorithm enters this step only when the cycle ratio of the previous largest cycle is decreased, i.e., $\tau(c^*) < \tau^*(G)$ in G'. In one case, c^* might remain to be the largest cycle in G' and we need to perform a global search to verify that $\tau(c) \leq \tau(c^*)\ \forall c \in G'$. In the other case, another cycle can replace c^* and becomes the new largest cycle in G'. Since we have no clue where this largest cycle is located, a global search for all cycles in G' is also required.

We leverage KO to perform this downward global search. In particular, the T_{v_s} we calculated during the cycle detection can be reused here. Thus, instead of running KO starting from a very large τ with the initial longest path tree as shown in Fig. 2(a), we can start KO from $\tau^*(G)$ with T_{v_s}. However, T_{v_s} is a longest path tree in $G^v_{\tau_0}$ rooted at node v, while the original KO requires the longest path tree rooted at an artificial node s, as shown in Fig. 2. Simply starting KO from $\tau^*(G)$ with T_{v_s} will make all cycles $c \in C_v$ cannot be examined, and the algorithm will be incorrect if $c^{*\prime} \in C_v$. Therefore, we modify KO like this: we add a pseudo edge (v_t, v_s) which is connecting node v_t to node v_s, and insert (v_t, v_s) into the max heap with $key(v_t, v_s) = d(v_t)/t(v_t)$. If (v_t, v_s) is picked during the execution of KO, it means $c^{*\prime} \in C_v$. Since $d(v_t)$ represents the largest cycle in C_v, it is safe for us to exit the MCR algorithm and output $\tau^*(G') = d(v_t)/t(v_t)$.

4. TIMING-DRIVEN DETAILED PLACEMENT

We propose a timing-driven detailed placement approach which applies our incremental MCR algorithm. Considering the type of circuits, i.e. asynchronous circuits [3] or synchronous circuits using retiming or clock scheduling techniques [2], whose performance is bounded by the MCR of the most critical cycle (c^*) in the circuit. For asynchronous circuits, c^* is defined as the timing loop which has the largest cycle delay divided by the number of tokens along the cycle. For synchronous circuits, c^* is defined as the timing loop which has the largest cycle delay divided by the number of flip-flops along the cycle. Here, we assume delay is proportional to the wirelength. Given an initial legalized placement, our goal is to reduce the MCR of the circuit by sequentially swapping a cell on c^* with one which is not on c^*.

The basic idea of our approach is illustrated in Fig. 8. First, we randomly pick a cell on c^*, i.e. cell v in Fig. 8. Next, we find the two neighboring cells of v on c^*, i.e. cell v_1 and v_2. The coordinates of v_1 and v_2 can define an optimal region $(x(v_1), x(v_2), y(v_1), y(v_2))$ for v, shown as the blue rectangle in Fig. 7. Assuming v' is a cell within this optimal region, by swapping v with v', we can minimize the total Manhattan distance of (v_1, v) and (v, v_2). Thus, $\tau(c^*)$ is reduced. However, it is possible that some cycle passing through v' becomes worse than c^* after the swap. Hence, we need to perform timing analysis using the incremental MCR algorithm to see whether the swap is beneficial before actually swapping the two cells. The details of our approach is shown in Algorithm 2.

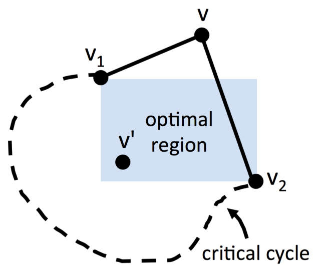


Figure 8: Timing-driven detailed placement.

5. EXPERIMENTS

The proposed incremental mean cycle algorithm is implemented in C++ and runs on a Linux PC with 94 GB of memory and 2.67 GHz Intel Xeon CPU.

Algorithm 2 A timing-driven detailed placement approach

```
Ensure: Reduce MCR of the circuit
 1: $n = 1$; /* loop counter */
 2: $best\_MCR = +\infty$;
 3: while $n < limit$ do
 4:     Randomly pick a cell $v$ on $c^*$ with neighboring cells $v_1$, $v_2$;
 5:     Set $optimal\_region := (x(v_1), x(v_2), y(v_1), y(v_2))$;
 6:     Set $x_{opt} := 0.5 * (x(v_1) + x(v_2))$;
 7:     Set $y_{opt} := 0.5 * (y(v_1) + y(v_2))$;
 8:     Move $v$ to $(x_{opt}, y_{opt})$.
 9:     Incrementally calculate MCR;
10:     for each $v'$ in $optimal\_region$ do
11:         Move $v'$ to $(x_v, y_v)$;
12:         Incrementally calculate MCR as $current\_MCR$;
13:         if $current\_MCR < best\_MCR$ then
14:             $best\_MCR = current\_MCR$;
15:             $best\_node = v'$;
16:         end if
17:     end for
18:     Move $v$ to $(x_{best\_node}, y_{best\_node})$;
19:     Incrementally calculate MCR;
20:     Move $best\_node$ to $(x_v, y_v)$;
21:     Incrementally calculate MCR as $best\_MCR$;
22:     $n = n + 1$;
23: end while
```

We generate a set of random graphs following the same graph size and method used in [4]. Given an input total number of nodes and total number of edges, we first generate the desired number of nodes in the graph. Next, we randomly pick two nodes in the graph and connect them. This step is repeated until the desired number of edges is reached. The self loops (an edge connecting a node to itself) and duplicated edges (two edges connecting the same pair of nodes) are disallowed. In addition, we connect all the nodes using a circle to make the graph strongly connected. Both the cost and transition of each edge is randomly generated between 1 and 300.

In the beginning (i.e., before any incremental changes is made), our algorithm uses KO to find the initial MCR and the corresponding largest cycle as a starting point. For all the random graphs and circuit benchmarks, our algorithm sets both the total number of negative and positive buckets to be 10^6. In addition, we set d_u to be 10. Thus, a distance range $[-10^7, +10^7]$ is covered, which is more than enough. In case any node has a distance below or above this range, we will assign it to the first or last bucket. We implement three other MCR algorithms for comparison: the linear programming (LP) approach, HOW and KO. The LP is formulated as we discussed in Section II-A, and solved using the API of Gurobi optimizer [18]. Both HOW and KO are implemented following the description in [1] [4]. In particular, we implement a binary heap as the max heap used in KO.

For each random graph, we sequentially perform 100 node changes and calculate the MCR after each change. For each changing node, we randomly change the costs of all its input and output edges. Two different methods are used to pick the changing node. In one method, which we referred as "M1", we randomly pick a node among all the nodes in the graph. Our algorithm is able to run faster in this case, as only the upward search might be performed if we are not changing c^*, which is quite often in M1. In another method, which we referred as "M2", we always pick a node on c^* to

Table I. Comparison on random graphs

Graph	# of nodes	# of edges	MCR			M1 Runtime (s)				M2 Runtime (s)			
			Init	M1	M2	LP	HOW	KO	Ours	LP	HOW	KO	Ours
r01	12752	36681	4.03	4.03	3.76	566.43	4.07	1.92	0.44	720.50	3.97	2.58	1.15
r02	19601	61829	4.39	4.39	4.06	2238.94	4.69	3.12	0.74	2885.36	9.92	4.79	2.06
r03	23136	66429	5.04	5.04	3.81	2380.95	3.20	2.72	0.92	3130.45	8.26	5.07	2.32
r04	27507	74138	4.50	4.50	3.61	2963.66	3.65	3.20	0.94	3866.92	12.48	5.16	2.77
r05	29347	98793	4.39	4.49	4.18	5090.83	18.03	8.07	1.42	4644.15	20.34	10.28	4.12
r06	32498	93493	3.87	3.87	3.56	6829.63	14.60	7.50	1.42	9060.00	24.86	9.35	4.02
r07	45926	127774	4.18	4.18	3.58	12377.20	9.95	6.93	1.80	15743.50	33.72	13.38	5.57
r08	51309	154644	4.84	4.84	4.11	–	10.01	7.52	2.11	–	27.24	11.40	6.31
r09	53395	161430	4.70	4.70	4.18	–	9.19	8.28	2.12	–	41.73	15.24	6.42
Norm.						–	6.496	4.136	1.000	–	5.253	2.223	1.000
r10	69429	223090	4.45	4.50	4.36	–	70.51	24.41	4.09	–	66.74	23.96	9.78
r11	70558	199694	3.84	3.84	3.57	–	95.00	24.77	4.20	–	95.62	24.42	9.53
r12	71076	241135	4.83	4.83	4.47	–	109.14	24.35	4.77	–	91.43	24.91	10.57
r13	84199	257788	4.79	4.79	4.07	–	29.65	17.12	4.25	–	100.52	23.84	10.80
r14	154605	394497	5.08	5.08	3.48	–	32.52	18.99	5.43	–	103.43	30.80	17.90
r15	161570	529562	6.30	6.30	4.45	–	22.46	22.59	7.21	–	268.69	52.77	26.33
r16	183484	589253	4.76	4.76	4.36	–	187.87	42.79	8.44	–	315.99	66.09	28.64
r17	185495	671174	5.24	5.24	4.94	–	488.10	74.59	12.75	–	620.42	98.54	42.62
r18	210613	618020	4.34	4.34	4.17	–	390.57	64.55	11.27	–	442.07	82.08	36.02
Norm.						–	22.848	5.034	1.000	–	10.952	2.224	1.000
r19	262144	851968	5.14	5.14	4.38	–	90.15	63.92	13.36	–	481.11	115.48	46.68
r20	311744	1013166	6.03	6.03	4.48	–	65.41	62.41	16.18	–	705.11	134.60	55.87
r21	370728	1204865	4.65	4.65	4.41	–	341.74	125.63	25.92	–	1023.05	169.74	65.69
r22	440879	1432834	5.06	5.06	4.65	–	399.31	154.68	36.86	–	1329.38	178.20	76.65
r23	524288	1703936	6.04	6.04	4.50	–	268.13	149.38	48.55	–	1112.68	240.61	97.21
r24	623487	2026333	4.59	4.59	4.44	–	1868.32	327.39	49.96	–	1679.11	305.82	116.13
r25	741455	2409729	29.75	29.75	43.13	–	71.00	47.10	38.21	–	75.93	53.94	91.21
r26	881744	2865667	5.52	5.52	4.66	–	294.65	189.08	44.39	–	1222.89	297.44	141.97
r27	1048576	3407872	4.78	4.78	4.57	–	1217.15	332.73	54.74	–	3355.29	450.52	180.88
Norm.						–	14.066	4.426	1.000	–	12.593	2.231	1.000

Table II. Analysis on medium and large size random graphs

Graph	Updates per node		M2 Runtime (s)		DB Runtime Breakdown (s)			
	BFS	DB	BFS	DB	CD	LUS	GDS	Others
r10	9.00	1.23	27.03	9.78	3.53	1.84	4.03	0.37
r11	9.58	1.32	22.17	9.53	3.65	1.00	4.53	0.35
r12	9.21	1.35	24.96	10.57	4.27	1.36	4.59	0.35
r13	9.27	1.21	28.89	10.80	3.84	1.30	5.31	0.35
r14	6.27	1.09	40.13	17.90	6.05	2.27	9.00	0.58
r15	9.75	1.19	72.70	26.33	8.39	3.78	13.48	0.66
r16	9.68	1.16	77.73	28.64	9.98	3.87	13.80	0.99
r17	10.14	1.16	89.75	42.62	13.09	5.65	22.49	1.39
r18	9.74	1.16	90.33	36.02	12.17	6.15	15.96	1.74
Norm.	7.614	1.000	2.465	1.000	0.338	0.142	0.485	0.035
r19	9.98	1.24	132.18	46.68	15.89	6.19	23.17	1.43
r20	10.71	1.21	175.30	55.87	19.02	8.25	27.04	1.56
r21	11.18	1.27	199.96	65.69	23.51	8.58	31.33	2.27
r22	11.22	1.15	232.59	76.65	24.71	8.71	40.64	2.58
r23	10.32	1.19	268.26	97.21	34.86	11.28	48.30	2.77
r24	10.99	1.29	381.76	116.13	42.58	15.95	52.21	5.40
r25	3.17	1.00	205.87	91.21	37.04	20.43	30.83	2.91
r26	8.85	1.05	439.19	141.97	44.63	21.62	70.85	4.87
r27	11.07	1.20	779.65	180.88	62.75	23.76	87.36	7.01
Norm.	8.257	1.000	3.227	1.000	0.350	0.143	0.472	0.035

Table III. Comparison on ISPD 2005 benchmarks

Design	# of nodes	# of nets	c^* moves	Skip moves	Total moves	HPWL x 10^6 (nm)		MCR		Runtime (s)		
						Init	Final	Init	Final	HOW	KO	Ours
adaptec1	210861	644176	39	2	514	77.78	77.89	2582.00	1390.60	248.42	409.98	48.19
adaptec2	254425	731135	23	4	398	87.68	87.73	4301.60	3456.80	80.10	49.43	34.40
adaptec3	450642	1289483	18	7	280	203.33	203.35	8084.50	2523.00	139.44	130.75	35.13
adaptec4	494590	1246535	24	1	482	183.59	183.72	2729.00	2115.00	327.05	171.16	95.88
bigblue1	277022	794445	24	2	225	97.21	97.30	3125.67	2973.00	40.99	30.59	32.13
bigblue2	528704	1267929	35	2	422	147.80	147.81	3494.91	3491.91	305.14	104.46	98.49
bigblue3	1094904	2511616	26	6	413	324.91	325.05	6020.33	3692.25	305.37	261.76	178.60
bigblue4	2168351	5691264	29	4	448	790.65	790.85	6085.50	3885.25	1128.32	704.68	403.59
Norm.			0.069	0.009	1.000	1.000	1.000	1.000	0.646	2.779	2.011	1.000

change. This is the most difficult case for our algorithm, as both the downward and upward search might be performed.

Table I shows the experimental results for random graphs, which are divided into three sets to simulate the applications with different scale. Columns "Init", "M1" and "M2" reports the initial MCR, the final MCR after 100 node changes using M1, and the final MCR after 100 node changes using M2 respectively. In general, LP is much slower than other algorithms. We denote the runtime of LP as "–", if it exceeds our runtime limit. Compared with HOW, our algorithm is about 5X∼23X faster among all the experiments. The performance of HOW is not good especially on large size graphs. Compared with KO, our algorithm is about 2X∼5X faster among all the experiments. In addition, as expected, the runtime of our algorithm in M1 is better than the runtime in M2.

Table II shows analysis of our algorithm in M2 on medium and large size graphs. We compare our MCR algorithm using the distance bucket approach (DB) with our MCR algorithm using the BFS approach described in [16]. The column "Updates per node" shows the average number of distance updates per node, and is calculated as (total # of node distance updates)/(# of T_{v_s} × # of nodes in the graph). It shows the DB approach is effectively reducing the updates per node and thus can achieve faster runtime. In addition, we show a runtime break down of our algorithm using DB. The columns "CD", "LUS" and "GDS" denote the cycle detection, local upward search and global downward search process respectively.

We use ISPD 2005 benchmarks [15] as the circuit benchmarks. Assuming a hypernet is connecting one output pin and p input pins of some gates, we represent this hypernet with p two-pin nets, by connecting the output pin with each input pin. Since there is no cell library type information in [15], we cannot calculate the wire delay. Instead, we set the cost of each two-pin net to be its HPWL, and the transition time of each two-pin net to 1. In addition, we ignore the fixed cells (i.e. terminals, macro blocks) and the nets connecting to them.

We stop the detailed placement if the improvement of MCR is less than 0.1% when we do the swap. Since our algorithm needs to incrementally calculate MCR at each move, a swap operation will need two calculations, while it only needs one calculation for HOW and KO. Thus, line 9 and line 19 in Algorithm 2 is required for our algorithm, but it is not needed for HOW and KO. Therefore, the total # of MCR calculations for our algorithm is larger than HOW and KO in this application. Table III shows the experimental results on circuit benchmarks. Columns "c^* moves", "Skip moves" and "Total moves" denote the # of moves on c^*, the # of moves skipped using our cost shifting idea and the total # of moves respectively. The runtime of the proposed detailed placement approach using three different MCR algorithms for timing analysis is compared. In particular, the placer based on our MCR algorithm is about 2X faster than the KO version and 2.8X faster than the HOW version.

6. CONCLUSIONS

In this paper, we have proposed an incremental MCR algorithm. The previous MCR allows us to break the graph at the changed node, and therefore detecting the changing directions of the MCR by solving a longest path problem in a graph without positive cycle. Based on the detected direction, our algorithm will either search upward or downward until the new MCR is found. We preform experiments on both random graphs and circuit benchmarks. The results show our algorithm is more efficient compared with HOW and KO.

7. REFERENCES

[1] A. Dasdan, S. Irani, and R. K. Gupta, "An Experimental Study of Minimum Mean Cycle Algorithms," *Tech. rep. 98-32, UC Irvine*, 1998.

[2] A. P. Hurst, P. Chong, and A. Kuehlmann, "Physical Placement Driven by Sequential Timing Analysis," in *ICCAD 2004*, pp. 379–386.

[3] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[4] A. Dasdan, "Experimental Analysis of The Fastest Optimum Cycle Ratio and Mean Algorithms," *TODAES*, vol. 9, no. 4, pp. 385–418, 2004.

[5] P. Min, N. Viswanathan, and C. Chu, "An Efficient and Effective Detailed Placement Algorithm," in *ICCAD 2005*, pp. 48–55, Nov 2005.

[6] G. Wu and C. Chu, "Detailed Placement Algorithm for VLSI Design with Double-Row Height Standard Cells," *TCAD*, no. 99, pp. 1–1, 2015.

[7] G. Wu, A. Sharma, and C. Chu, "Gate Sizing and Vth Assignment for Asynchronous Circuits Using Lagrangian Relaxation," in *ASYNC*, pp. 53–60, 2015.

[8] L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck, "An Experimental Study of Minimum Mean Cycle Algorithms," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 1–13, Society for Industrial and Applied Mathematics, 2009.

[9] J. Magott, "Performance Evaluation of Concurrent Systems using Petri Nets," *Information Processing Letters*, vol. 18, no. 1, pp. 7–13, 1984.

[10] R. M. Karp and J. B. Orlin, "Parametric Shortest Path Algorithms with An Application to Cyclic Staffing," *Discrete Applied Mathematics*, vol. 3, no. 1, pp. 37–45, 1981.

[11] N. E. Young, R. E. Tarjant, and J. B. Orlin, "Faster Parametric Shortest Path and Minimum-balance Algorithms," *Networks*, vol. 21, no. 2, pp. 205–221, 1991.

[12] R. A. Howard, "Dynamic Programming And Markov Process," *The M.I.T Press, Cambridge, Mass*, 1960.

[13] N. Chandrachoodan, S. S. Bhattacharyya, and K. Liu, "Adaptive Negative Cycle Detection in Dynamic Graphs," in *ISCAS 2001.*

[14] E. L. Lawler, "Combinatorial Optimization: Networks and Matroids ," *Holt, Rinehart and Winston, New York*, 1976.

[15] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ISPD2005 Placement Contest and Benchmark Suite," in *ISPD 2005*.

[16] R. E. Tarjan, "Shortest Paths," *Tech reports*, 1981.

[17] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest Paths Algorithms: Theory and Experimental Evaluation," *Mathematical programming*, vol. 73, no. 2, pp. 129–174, 1996.

[18] Gurobi Optimizer: http://www.gurobi.com.