基于 Polish 表达式的 Slicing 布图规划报告

李依林 2016011502　　周沁泓 2016010493

January 20, 2019

目录

第 1 章. 布图规划综述

布图规划 (Floorplanning) 问题接受一组或硬或软的模块 (hard/soft blocks) 及网表 (netlist)。目标是确定每个模块的位置, 使它们互不重叠。对于软模块, 还要确定具体尺寸。传统的优化目标包括总面积, 线长, 有时还需要考虑热功耗 (thermal hotspot), 电源噪声 (power supply noise) 等。实际布图规划还面临固定轮廓, 固定模块等限制。

布图规划根据拓扑性质, 可以分为切分结构 (Slicing Floorplanning) 和非切分结构 (Non-Slicing Floorplanning)。切分结构要求布图规划能够递归地进行水平或垂直切分 (h-cut / v-cut) 得到, 而非切分结构无此限制。非切分结构的描述能力更强, 例如 wheel 是最简单的非切分布图。

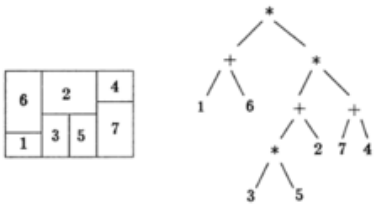

Figure 1.1: Slicing Tree

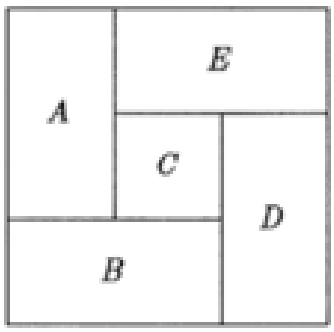

Figure 1.2: Wheel

布图规划一种思路是将布图规划对应于某个表示, 将所有合法的表示作为解空间, 使用模拟退火等方法或基于划分的方法求解。切分结构经典的表示方法包括 Slicing Tree 及 Normalized Polish Expression. 非切分结构的表示方法有 O-Tree, B*-Tree, Sequence-Pair 等。另一思路是将布图规划转化为整数线性规划问题 ILP (Integer Linear Programming) 求解。本文主要讨论了基于 Normalized Polish Expression (NPE) 的布局规划算法, 并将其与 Sequence-Pair (SP) 作了对比。由于后者并非此项目的重点，我们将不会具体论述其基本原理，而只在工作框架一节中给出一些关键之处。

第 2 章. Normalized Polish Expression (NPE)

每一个划分自然对应一棵 Slicing Tree, 其叶节点对应一个不可划分的模块 (leaf block); 非叶节点对应一个切分操作 (h-cut / v-cut), 以它为根的子树对应一个组合模块 (composite block)。每一棵 Slicing Tree 的后序遍历给出了一个以操作数 (模块) 和操作符 (, 其中对应 h-cut, 对应 v-cut) 为符号的 Polish Expression。它满足如下性质

1. 长度为 $2N-1$(N 为模块数), 且每个模块恰好出现 1 次
2. (The balloting property) 任意非空前缀中, 模块数大于操作符数
   但是从 Polish Expression 不一定能确定唯一的 Slicing Tree. Wong 和 Liu 证明了, 当 Polish Expression 当满足
3. 相邻操作符不相同时, 对应唯一的 Slicing Tree. 此时称该表达式为 Normalized Polish Expression (NPE).
   作者给出了 3 种变动 NPE 的方法
   - M1: 交换相邻的某两个操作数
   - M2: 某条操作符链取反
   - M3: 交换相邻某两个操作数和操作符

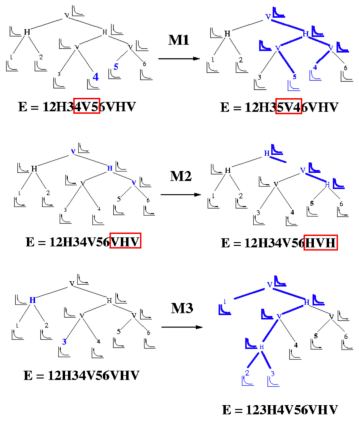

Figure 2.1: M1, M2, M3 操作

其中 M1, M2 总是可以实施。M3 在交换 operator e_i 与 operand e_{i+1} 时也总是可以实施; 而交换 operand e_i 与 operator e_{i+1} 可以实施, 当且仅当 $2N_{i+1} < i$, 其中 $N - k$ 代表表达式长 k 的前缀中操作符的数量。

值得注意的是, 每次变动最多只会带来两个树路径自底向上的更新, 因此可以增量式地快速更新评价函数 (incremental evaluation of cost function)。如果每个模块都代表一个确定方向的矩形, 那么不难看出 M1, M2, M3 的时间复杂度都是 $O(h)$, h 为树的高度。由于实际操作中通过模拟退火的方式进行这几种操作, 有一定的随机性, 时间复杂度应该是 $expectedO(logn)$ 的。相比之下，如果不采取增量更新，时间复杂度为 $O(n)$.

前面讨论的 M1, M2, M3 暂时只限于不可旋转的硬矩形模块。类似地, Slicing Tree 和 NPE 也可以推广到允许 L 形模块的情况。但我们的项目中只考虑矩形硬模块及软模块。为了至少解决可旋转的硬模块的问题 (它是软模块的一个特例), 一种方法是加入新的交换动作 M4

- M4: 将某个操作数旋转 90°

为简便, 我们将这种策略称为 polish.

我们将另一种实现了的策略称为 polish-curve, 它能够处理一般的软模块, 并且对解空间有良好的压缩效果, 如下所述。

第 3 章. Shape Curve

对于或硬或软的矩形模块, 可以用 shape curve 统一描述。

宽为 w, 高为 h 的硬模块, 不可旋转。$x \geq w, y \geq h$ 宽为 w, 高为 h 的硬模块, 可旋转 90°

对于满足

$$\begin{cases} x \geq w & (3.1) \\ y \geq h & (3.2) \end{cases}$$

or

$$\begin{cases} x \geq w & (3.3) \\ y \geq h & (3.4) \end{cases}$$

面积为 a, 长宽比不超过 $k(k \geq 1)$ 的软模块,

$$\begin{cases} x \cdot y \geq a & (3.5) \\ k^{-1} \leq \dfrac{y}{x} \leq k & (3.6) \\ x, y0 & (3.7) \end{cases}$$

上述平面规划的边界就是 shape curve。

对于 shape curveΓ,Λ, 定义 $\Gamma \times \Lambda = (u+v, w)|(u, w) \in \Gamma(v, w) \in \Lambda$ 对于以 Γ, Λ 为 shape curve 的两个模块 x, y, $x+y$(垂直放置) 的 shape curve 是 $\Gamma + \Lambda$, $x \times y$ (水平放置) 的 shape curve 是 $\Gamma \times \Lambda$, 反之, 确定了 $x+y$ 或 $x \times y$ 的合法尺寸, 也可给出合法的尺寸。

通常而言, 一个比较合理的 curve shape 应能用参数

$$\begin{cases} x = u(t) & (3.8) \\ y = v(t) & (3.9) \end{cases}$$

表示, 其中 $u(t), v(t)$ 是连续函数, $u(t)$ 非严格递增, $v(t)$ 非严格递减。不难证明, 这样的 curve shape 对于 + 和 × 操作是封闭的。

进一步考虑多个可旋转 90° 的硬模块的 shape curve 经过有限次 + 和 *times* 操作复合形成的情形。不难看出, 它由首尾相接的铅垂线和水平线构成。对于这样的 shape curve, 只需要记录每个铅垂线下端点和水平线左端点相接的点集即

可代表整条曲线。对于一般的 shape curve, 该方法仍然适用。选取曲线上足够多的点, 这些点对应的铅锤线-···-水平线可逼近原曲线。

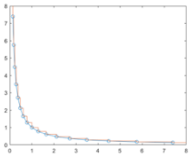

Figure 3.1: 用铅垂线和水平线逼近 $y = x^{-1}$

使用上面所述的方法, 每个叶节点 (模块) 含有 2 个或 1 个点。假设某个内部节点左子的 shape curve 含有 m_1 个点, 右子含有 m_2 个点, 则合并后该节点的点数 m 满足

$$max(m_1, m_2) \leq m \leq m_1 + m_2$$

时间复杂度为 $O(m_1 + m_2) = O(2m)$

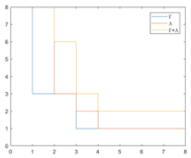

Figure 3.2: shape curve “+” 操作

将该时间复杂度运用到平衡的 Slicing Tree 上, 可以用下式估算从一个叶节点从第向上更新的时间复杂度

$$T = 1 + 2 + 4 + ... + 2^h, whereh = O(logn)$$

故而该情形下复杂度为 $O(n)$.

实际模拟退火过程中，树的平衡状态通常不会太差，并且 M1, M2, M3 操作最多从两个叶节点从底向上更新，故而 polish-curve 策略 M1, M2, M3 操作的时间复杂度为 $expectedO(n)$. 相比之下, 不使用增量更新 M1, M2, M3 操作的时间复杂度为 $expectedO(nlogn)$.

相比于 polish 策略，polish-curve 策略的处理能力更强，并且将解空间压缩到大约 $\frac{1}{2^m}$ (每个矩形模块的两种可能取向, 假定长宽不等), 付出代价是每次操作的时间复杂度有所提高。

第 4 章. 功能定义

运行 bin/main 程序，从标准输入或文件读取 YAL (Yet Another Language) 格式文件，可以选择 polish, polish-curve, lcs, dag 四种策略 (详见下文 NPE Solver, SP Solver 部分), 将经过验证的布图结果输出到标准输出（stdout）或指定文件，将运行信息 (布图空间利用率、运行时间等) 输出到标准错误（stderr）。

运行 src/visualize/visualize.py, 以 bin/main 输出文件作为输入, 可视化布图结果。

从 YAL 解析的有效信息有且仅有各模块的长和宽, 以及顶层模块 (YAL PARENT) 所给出的”元件例化”信息, 即每个定义的模块被实例化的次数。求一种布图方案, 使得

1. 任意两模块不重叠
2. 使所有模块的矩形包络面积最小, 不考虑线长因素

4.1 关于不考虑线长的说明

有必要说明我们不考虑线长的原因，以及该情形下使用上述求解框架的合理性。如前面所述，polish 策略单次更新的时间复杂度是 $O(h)$, polish-curve 策略为 $expectedO(n)$; 对于 Sequence-Pair, LCS 策略单次更新的时间复杂度为 $O(nlogn)$, 使用 DAG 策略为 $O(n^2)$. 设连线数量为 $m = O(n^2)$, 则如果简单在每次更新后根据布局结果计算线长的时间复杂度分别变为 $O(n+m), expectedO(n+m), O(nlogn+m), O(n^2)$. 即便进一步假定 $O(n)$，polish 策略时间复杂度的退化也是明显的。然而相比于更新 Slicing Tree 节点的几何形状或者 shape curve，快速地增量式更新总线长, 无论是对于 Slicing Tree 的结构还是 Sequcence-Pair 结构都并不容易。有鉴于此, 为了简洁性并突出各策略的差异，我们并未考虑线长。尽管如此, 只要能够承受额外的时间开销 (使得部分策略渐进时间复杂度退化), 加入线长也是容易的。

读者可能会问，既然不考虑线长，为何不使用 Stockmeyer 算法，确定性地求解使 slicing 结构总面积最小的模块取向。的确，只限定于矩形模块，不考虑线长，

这是更简单也更高效的做法。但我们实现的 polish 和 polish-curve 两种策略更为一般化：1. 如前文所述，它们能方便地加上对线长因素的考虑，但 Stockmeyer 算法无法胜任 2. polish-curve 策略可接受软模块作为输入。

第 5 章. 项目结构

5.1 YAL Parser

位于 src/yal 文件夹下, yal 命名空间中。

使用 Flex 和 Bison 工具，对 YAL 进行 LALR 分析，并生成对应抽象语法树。由于 YAL 比较简单, 语法树只是列表之类的结构。

5.2 NPE Solver

位于 src/polish 文件夹下, polish 命名空间中。

该模块在逻辑上又可以分为两部分: Slicing Tree 数据结构以及模拟退火算法。

5.3 Slicing Tree

slicing_tree 被定义为一个类似 STL container 的模板类，以 polish 和 polish-curve 所对应节点类型为 value_type, 使用 Allocator (C++ concept) 来管理内存. slicing_tree 定义了 polish 和 polish-curve 策略共有的方法, 包括 M1, M2, M3 操作, 后序遍历顺序的双向迭代器 (bidirectional iterator), 以及由 YAL 模块列表随机初始化树的操作。

slicing_tree 派生出两个子类 (也是模板类), polish_tree 以及 vectorized_polish_tree, 分别对应于 polish 和 polish-curve 策略。二者补充定义了两种策略差异较大的 floorplan (即确定模块位置和取向) 方法。前者还增加了 M4 操作。

在模拟退火过程中，首先调用 slicing_tree::consctuct 随机初始化树, 然后对树进行一次后序遍历将所有迭代器 (或者说节点指针) 存在一个 vector 中, 于是这个 vector 就隐式地代表了树对应的 NPE. 然后用 slicing_tree::rotate_nodes (M1, M3),slicing_tree::invert_chain(M2) (如果是 polish 策略还有 polish_tree::rotate_leaf(M4)) 进行模拟退火所需要的操作，在对树操作的同时还要同步地交换 vector 中迭代器的顺序，使得 vector 的内容始终与树后序遍历保持一致。额外维护 vector 的作用在于, 为树提供了后序遍历意义上的 O(1) random access, 从而维持了几种

操作原先的时间界。(注: 实际上我们尚未论证为找到一个合法操作进行随机尝试的时间, 在模拟退火算法部分会讨论这个问题)

大多数上述模板类都带有一个 Allocator 模板参数, 其作用是能够在最终代码中将 std::allocator 替换为某些特殊的 allocator (我们使用的是 boost::fast_pool_allocator), 从而减小内存分配 / 释放的开销。应该说, 最终对于 polish 和 polish-curve 策略内存分配 / 释放并非一个明显的瓶颈, 因为内存分配 / 释放只在以下几种情况下发生:

1. 随机生成树
2. 将当前树拷贝到最优树
3. polish-curve 策略更新 shape curve, 并且存储 shape curve 的 vector 容量不足
4. 每轮模拟退火开始和结束时 SA 模块与其调用者间拷贝树

5.4 模拟退火算法

5.5 整体流程

模拟退火的整体流程如下图所示。每一次模拟退火中

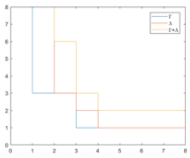

Figure 5.1: 模拟退火流程

- 首先设定初始温度和初始状态。
- 在每一个温度下，不断的在邻域中寻找一个新解。按照 Metropolis 准则接受新解，即若新解的目标函数小于旧解，则接受；若新解的目标函数大于等于旧解，则接受概率为 $e^{-\frac{\Delta f}{T}}$ 。直到达到该温度下的平衡条件。
- 按照降温策略降低温度。若未达到模拟退火终止条件，重复上一个操作。若已经达到终止条件，则这一次模拟退火终止
- 返回在整个过程中记录下的最优解

在本项目中，为了使结果稳定在一个较好的水平之上，我们不断地重复上述操作，进行多次的模拟退火，每次模拟退火都是从上一次模拟退火的最优解开始，完整地进行一遍模拟退火流程。

5.5.1 邻域构造方法

本项目中的四种邻域构造方法总结如下, polish 策略使用 M1 –M4, polish-curve 策略使用 M1-M3

- M1: 交换相邻的某两个操作数
- M2: 某条操作符链取反
- M3: 交换相邻某两个操作数和操作符
- M4: 将某个操作数旋转 90°

其中，在第三种邻域构造方法中，有可能会产生非法的操作。例如在 “12+” 中，若交换 “2” 操作数和 “+” 操作符，那么 Polish 表达式将会变成 “1+2”，这显然是非法的。为了高效地处理这个问题，我们在 Slicing Tree 试图进行 M3 操作时进行检查 (如果不合法该操作将是 no-op 并返回 false)，而不是按照前缀中操作数和操作符数量来检查。随机重复该操作，直到 M3 交换位置合法为止。这样只需要 $expectedO(h)$ 时间便能找到一个合法的 M3 操作, 不会改变操作本身的时间复杂度。(注: 另一种可行的方法是维护一个树状数组来维护前缀和, 其复杂度为 $O(logn)$)

5.5.2 初温、平衡条件与终止条件等参数选择

- 初始温度：为了让模拟退火开始时，对于邻域中的新解有一个较高的接受概率 P，我们采用 $T_0 = \frac{\overline{\Delta}}{p}$，其中 $\overline{\Delta}$ 表示初始状态邻域解的目标函数与初始状态的目标函数差值的平均值。我们在初始状态的邻域中随机抽取一定数量的新解，计算其目标函数与初始状态目标函数差值的平均值，从而得到 $\overline{\Delta}$
- 降温策略：$T_k = k_T \cdot T_{k-1} - \Delta_T$
- 平衡条件：该温度下接受解的次数达到模块数量的一定比例
- 模拟退火结束条件：温度低于一定的阈值

5.6 SP Solver

位于 src/seqpair 文件夹下, seqpair 命名空间中。类似 polish 模块, 该模块也可以分为两部分: Sequence-Pair 表示以及模拟退火算法。

5.6.1 Sequence-Pair

Sequence-Pair 表示到实际布图的转化是该算法的核心。一种较早提出的方法是将 Sequence-Pair 转化为水平和垂直方向的约束图，将问题转化为有向无环图 (DAG) 最长路径问题, 其时间复杂度为 $O(n^2)$. 一种改进的方法是使用最长公共子序列 (LCS) 算法来求解, 其时间复杂度为 $O(nlogn)$. 对应于 LCS 和 DAG 两种求解方法, 分别定义了 LCSPackGenerator 和 DAGPackGenerator. 它们的主要功能是: 内部维护一个 SP 表示, 并且被调用时首先执行一次随机交换, 然后根据交换后的 SP 表示产生具体布图结果, 并返回长和宽。此外还定义了 rollback 方法用于撤销前一次交换 (用于模拟退火拒绝某次变化时), 以及 shuffle 方法对整个 SP 进行随机置换 (用于模拟退火根据能量样本标准差确定初始温度)。

实验结果表明，论文中提及的邻域选择方式效果并不好。实际上 SP 的表示和 TSP 问题的解类似，都是 $[0, n)$ 的一个排列。参考典型的 TSP 的模拟退火求解方式，我们测试了以下的邻域选取策略, 其中 1, 3 是论文中提出的 (记 Sequence-Pair 的两个向量为 x, y；略去了旋转 90° 的交换，它对于问题的 P-admissibility 是必要的):

1. 交换 x 中两个元素
2. 交换 y 中两个元素
3. 同时交换 x，y 中两对元素
4. 转置 x 的一个子序列
5. 转置 y 的一个子序列
6. 同时转置 x, y 的一对子序列
7. 将 x 的一个子序列循环左移
8. 将 y 的一个子序列循环左移
9. 将 x，y 的一对子序列循环左移

前期测试发现，综合使用后 6 种交换方式效果较好，而原先论文中提及的 1，3 会导致较低的空间利用率。

对于 LCS 策略, 实现中用到了 std::map, 并且频繁进行插入删除操作。在此处使用 boost::fast_pool_allocator 有很好的常数优化效果。

5.6.2 模拟退火算法

模拟退火进行单轮，通过随机 SP 对应布图能量的样本标准差以及给定的初始接受率确定初始温度，以接受率作为终止条件，等比例降温，并且在当前能量过高相比于已知最优解能量过高时进行 restart 操作（重新从最优解开始模拟退火）。

5.7 Visualization

位于 src/visualize 文件夹下。

用于可视化的 python 脚本，将 solver 输出的 (x, y, width, height) 格式的布图结果绘制出来。

5.8 其他

程序入口点定义在 src/main.cpp 中。

另外有一部分位于 src/aureliano 文件夹下, aureliano 命名空间中。主要是提供一些计时以及 SFINAE 的工具。

第 6 章. 测试结果展示

在如下环境中分别对三种算法对应的程序进行用时测试 每次测试均进行 5

操作系统	Windows 10 1803
处理器	Intel i7-6700HQ 2.60GHz

表 6.1: 测试环境

轮，取 Utility 和时间的中位数作为测试结果，Utility 测试结果记录如下（单位：%）

电路文件 算法	polish-curve	polish	lcs	dag
apte.yal	99.23	97.97	98.41	98.90
ami33.yal	93.03	85.92	91.21	88.53
ami49.yal	92.01	86.25	87.33	87.26

表 6.2: Utility 测试结果

可视化程序输出结果如下四图（以基于 Polish 表达式的模拟退火在 ami49.yal 上的表现为例）

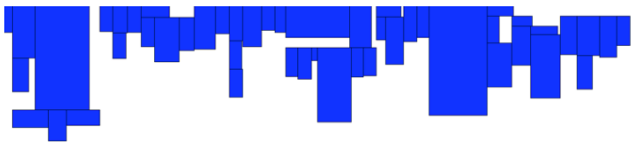

Figure 6.1: 第 100 次降温时最优解

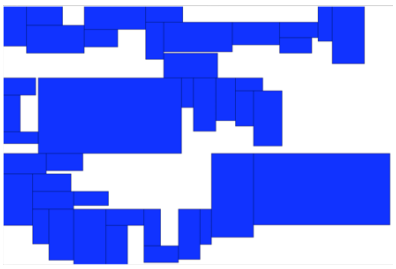

Figure 6.2: 第 200 次降温时最优解

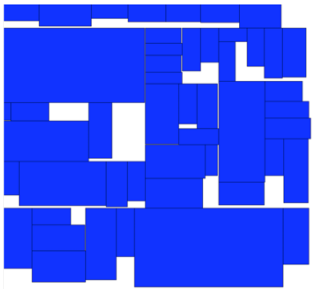

Figure 6.3: 第 600 次降温时最优解

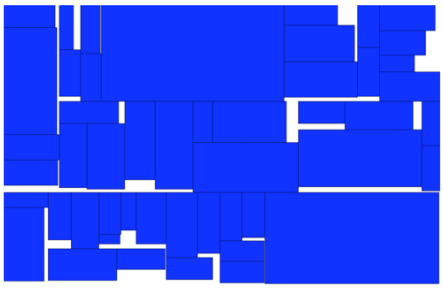

Figure 6.4: 第 3100 次降温时最优解

第 7 章. 结果分析

测试结果表明，polish 策略和 polish-curve 策略在大致相当的用时下，后者的空间利用率明显更高，可见 shape curve 对于解空间的压缩有良好效果。

由于 lcs 和 dag 策略的唯一区别是将 Sequence-Pair 表示转化为具体布图，二者空间利用率期望相等，只是运行时间有别。测试结果并未体现出二者单次操作时间复杂度的差异，这是 YAL 测试样例规模较小所致，由于 polish, polish-curve 和 lcs, dag 使用的模拟退火策略有所不同，它们之间空间利用率的差异还并不能完全归结于 NPE 与 Sequence-Pair 表示方法的不同。使用统一的模拟退火框架来处理和比较这四种不同策略，是本项目一个可能的后续工作。

尽管如此，我们注意到多轮模拟退火的确使空间利用率有较大提高（运行时输出可见每轮的最优空间利用率），并且时间仍在可接受范围之内，是提升布图效果的一个不错的选择；同时轮次可以根据对布图效果和运行时间的需求来选择，增加了灵活性。

我们实现了对 polish 策略和 polish-curve 策略单次操作后面积指标的增量更新，而尚未讨论总线长的问题。这两种策略以及 Sequence-Pair 表示情况下，如何对总线长进行高效的增量更新，是一个值得进一步研究的问题。

第 8 章. 參考資料

- Wong, D., Leong, H., Allen, J. (1988). Simulated Annealing for VLSI Design (Vol. 42). Boston: Springer.
- Wong, D., Liu, F. (1989). Floorplan design of VLSI circuits. Algorithmica, 4(1), 263-291.
- Cong, J., Romesis, M., Shinnerl, J. (2006). Fast floorplanning by look-ahead enabled recursive bipartitioning. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 25(9), 1719-1732.
- Lim, S. (2008). Practical Problems in VLSI Physical Design Automation. Dordrecht: Springer Netherlands.
- Y.-W. Chang, Floorplanning, Unit 5: Floorplanning, retrieved from
- http://cc.ee.ntu.edu.tw/ ywchang/Courses/EDA04/lec5.pdf
- MCNC Benchmark Netlists for Floorplanning and Placement, retrieved from
- https://s2.smu.edu/ manikas/Benchmarks/MCNC_Benchmark_Netlists.html
- H. Murata, K. Fujiyoshi, S. Nakatake and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 12, pp. 1518-1524, Dec. 1996.
- Xiaoping Tang, Ruiqi Tian and D. F. Wong, "Fast evaluation of Sequence-Pair in block placement by longest common subsequence computation," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 12, pp. 1406-1413, Dec. 2001.