本 科 学 位 论 文

电容提取的随机行走算法及空间管理技术的实现与优化

院　　系：　微电子学院

专　　业：　微电子学系

姓　　名：　谷年龙

指导教师：　严昌浩

完成日期：　2016 年 5 月 19 日

摘要

随着深亚微米超大规模集成电路(VLSI)的发展，金属线宽逐年下降，同时互连线的层数也随着芯片规模的增大逐渐增加，互连线寄生电容所引起的延时在总延时中所占的比重越来越大。因此，寻找能够快速准确地计算互连线电容的方法显得尤为重要。本文基于清华大学的喻文健教授的论文实现了采用随机行走算法提取互联电容的技术，该方法是一种基于数学统计的数值方法，特点是原理简单，易于实现，内存占用低，适用于大规模集成电路的互连电容提取。本文介绍了随机行走算法的基本原理，对随机行走算法实现过程中的关键问题，如高斯面的生成、权重函数的计算、随机行走的终止条件以及随机行走的并行加速等做了分析。此外，本论文重点讨论了空间管理技术的现状，介绍了三维数组和八叉树结构在空间管理中的应用。对八叉树结构的生成算法提出了优化意见，并且在原有空间管理技术的基础上提出了改进办法，有效加速了定位节点单元和生成最大转移立方体的过程。

关键词：电容提取　　随机行走法　　空间管理　　八叉树

Abstract

With the development of deep-submicron Very Large Scale Integration Circuits (VLSI), metal linewidth decreases while the number of interconnect layers gradually increases. The influence of the delay caused by interconnect capacitance becomes increasingly considerable. Therefore, it is important to find a method which can efficiently compute the interconnect capacitance with high accuracy. In this thesis we discuss a statistical numerical method called floating random walk (FRW). This method is simple in principles, easy to implement, and has a relatively low memory cost, which makes it suitable to capacitance extraction of VLSI interconnect layers. We first introduce basic principles of FRW method, then analyze main topics of the FRW process, such as the generation of Gaussian surface, weight value function, surface Green function, the termination condition of FRW, as well as the parallel implementation. Besides, in this thesis we focus on space management techniques and introduce several data structures, including 3-D array and octree. We improve the algorithm of octree generation and introduce an improvement of space management on the basis of the original grid-octree data structure, which speeds up the process of locating leaf cell and generating the largest transition cube remarkably.

Keywords: **Capacitance Extraction** **Floating Random Walk** **Space Management** **Octree**

目录

第一章 绪论

§1.1 研究背景和意义

集成电路发展早期，金属线宽较大，互连线的层数较少，互连线寄生参数引起的延时远小于门延时。随着深亚微米大规模集成电路（VLSI）的发展，金属线宽逐渐减小，同时由于芯片规模的扩大，互连线的总长度以及层数逐渐增加，这些因素导致互连线的 RC 延时在电容总延时中所占的比重越来越大。因此，在分析电路时序时必须充分考虑互连线的延时，这就要求精确计算互连线的寄生参数。其中互连线电容对电路的性能、延时以及噪声特性都有显著的影响，寻找能够快速精确地计算互连电容的方法成为近年来十分重要的课题。

早期电容提取的数值计算方法是基于边界元法[1]和有限元法[2]的确定性算法，这类算法的优点是速度快，计算结果精确；缺点是计算量较大并且对内存使用率较高，因此不适用于大规模集成电路互连电容提取。近年来，一种基于蒙特卡罗法的随机行走算法[3]-[7]逐渐发展，与传统的确定性算法相比，该算法原理简单，对版图的预处理较少，同时内存占用率低，因此适用于大规模集成电容的互连电容提取。本论文将介绍随机行走算法的现状、实现过程及改进方法。

§1.2 随机行走算法提取三维互连电容的研究情况

采用随机行走法进行电容提取的二维模型最早在 1992 年由 Y. Le Coz 和 R. B. Iverson 提出[3]，其基本思想是通过统计抽样的方法逐渐逼近高斯定理积分表达式的精确值。随机行走的过程如下：首先选取一个能够将导体完全包围的高斯面，然后从这个高斯面上按照一定的分布选择一个点P_0作为随机行走算法的起始点；以点P_0为中心，生成一个不包含任何导体的立方体C_1，然后按照一定的概率分布从立方体C_1的表面上选出一个点P_1。其中，从P_i到P_{i+1}的过程称为一次跳转；从转移立方体C_i的表面上选出下一个跳转点P_i服从的概率分布函数称为表面格林函数；跳转终止的条件是当P_i刚好处在某个导体的表面，或者P_i离整个版图的距离超过了一定的范围。根据数理统计的概念，一次随机行走可以得到导体上电荷量的一个样本，当进行 N 次随机行走之后得到的样本均值可以看作是对导体携带的电荷量的估计。根据电荷量和电容之间的关系，就可以计算出导体之间的电容或导体的自容。Y. Le Coz 等在此基础上提出了三维随机行走算法[4,6]。

目前，对随机行走算法的探讨和优化可以基本分成如下几个方向：

1、 减少随机行走结果的方差，使随机行走的结果更快收敛。

减少采用蒙特卡罗法所带来的方差，使随机行走过程更早地收敛，本质上是减少随机行走的次数。Nikhil Bansal 讨论了二维平面条件下减小方差的数学方法[5]，在 2005 年 S. H. Batterywala 提出若干技术减小随机行走过程中的方差[7]，清华大学的喻文健等主要讨论了通过重点采样和分区域采样的方法来减少权重函数表达式的方差[8]。

2、减少平均每次随机行走消耗的时间。

为减小平均每次随机行走消耗的时间，需要使每一次随机行走中包含的跳转的次数尽可能少，同时每次跳转的时间尽可能短，这就要求在每一次跳转时快速产生尽可能大的转移立方体。显然，以一个点为中心产生最大内部不含导体立方体的基本方法是检查该点到版图内所有导体的距离，但是这会消耗大量的时间，因此寻找高效的空间管理技术成为重要的研究方向。Nikhil Bansal 讨论了多种空间管理技术的数学模型，分析了多种空间分割方法的可行性，并指出八叉树结构(Octree)是一种比较有效的空间管理方法[5]。喻文健等比较系统的介绍了基于八叉树结构的空间管理技术，提出了若干提高管理效率的方法，如不完整候选导体列表、八叉树结构与三维数组相结合的手段等[9]。然而，在八叉树结构与三维数组的结合方式、定位叶子节点单元、处理随机行走出界等问题上仍需要进一步改进。本文也力图在前人成果的基础上，进一步对空间管理技术进行优化，提高随机行走的速度。

3、 随机行走的并行实现。

随机行走法提取电容的一个优点是其可以实现并行计算，因为不同行走之间是相互独立的过程，可以利用这一特性实现并行计算，充分利用多核处理器的计算能力，同时不会引起过多的内存占用。[8]给出了使用多核处理器环境下的加速效果，同时对空间管理的并行实现的可行性进行了探讨。

此外，可优化的方向还包括高斯面的生成[10]、表面格林函数与权重函数的离线计算、多种介质层下的随机行走问题[8,10]、随机行走法与边界元法协同计算等等。

§1.3 论文框架

本文主要介绍了采用随机行走算法提取三维互连电容的基本原理、实现方法，对随机行走算法并行实现中的问题进行了分析，探究了空间管理技术的实现方法并提出了几点改进方法。

第一章 绪论，介绍了随机行走算法提取电容的研究背景与意义，以及随机行走算法的研究现状，给出了论文的框架。

第二章 随机行走算法简介，介绍了电容提取和随机行走法的基本原理，并对空间管理技术的概念进行介绍。

第三章 空间管理技术的实现，介绍了空间管理中的重要定义和定理，引入了候选导体列表的概念，探讨了八叉树结构和三维数组以及组合型空间分割方法，并探讨了空间管理技术的加速方法。

第四章 随机行走算法的实现，介绍了随机行走算法的完整流程，并实现了流程中的关键步骤，对随机行走实现中可能遇到的问题进行了探讨。

第五章 实验结果，给出编程实现的测试结果。

第六章 总结与展望，对整个毕业设计的工作总结以及对将来研究方向的展望。

第二章 随机行走算法简介

§2.1 电容提取的基本原理[5]

首先给出电容提取的基本模型。假设版图中有两个导体 i 和 j，其电势分别为φ_i和φ_j，导体 i 的电荷量为q_i ,那么导体 i 和导体 j 之间的电容C_{ij}满足关系：

$$q_i = C_{ij}(\varphi_i - \varphi_j)$$

为简便，令 $\varphi_i = 0, \varphi_j = 1,\ \forall i \neq j$,那么则成立

$$q_i = -C_{ij}$$

因此可以把导体间电容的问题转化成导体携带的电荷量的问题。

当然上述表达式只是一个近似，实际上一个导体上携带的电荷量和空间内各个导体的电势都有关系，可以根据高斯定理准确计算导体 i 携带的电荷量：

$$q_i = \epsilon \oiint_G \vec{E}(\vec{r}) . \hat{n}(\vec{r}) d\vec{r} \tag{2.1.1}$$

其中ϵ为互连介质的介电常数，G表示包围导体的闭合高斯面，$\vec{r}$表示高斯面G上的任意一点，$\vec{E}(\vec{r})$表示在高斯面上的电场强度，$\hat{n}(\vec{r})$表示在高斯面上某一点$\vec{r}$方向指向高斯面外部的单位法向量。根据电压与电场之间的关系$\vec{E} = -\nabla\phi$, 代入式(2.1.1)可得

$$q_i = -\epsilon \oiint_G \nabla_r \phi(\vec{r}) . \hat{n}(\vec{r}) d\vec{r} \tag{2.1.2}$$

因此，求导体 i 上所带的电荷的问题转化成了求导体高斯面上任意一点$\vec{r}$处的电势$V(\vec{r})$的问题。在一个没有净电荷的空间内，电势满足泊松方程

$$\nabla_r . [\epsilon(\vec{r}) \nabla_r \phi(\vec{r})] = 0 \tag{2.1.3}$$

若考虑介电常数为恒定值，那么则成立拉普拉斯方程

$$\nabla_r^2 \phi = 0 \tag{2.1.4}$$

传统的确定性数值解法，如边界元法、有限元法的基本思想是通过解(2.1.4)所示的拉普拉斯方程计算高斯面上的电势。[5]中指出了这类方法的缺陷：（1）内存占用高。在求解拉普拉斯方程需要将三维空间栅格化，当版图的规模增加时，栅格点的数目会迅速增加，导致很大的内存占用。（2）在计算局部点的电势的时候需要进行全局计算，这导致大量的计算都消耗在不感兴趣的点的电势计算上。这两点缺陷导致此类方法不适用于大规模集成电路互连电容的提取。

§2.2 随机行走法的基本原理[5][8]

随机行走算法是一种基于蒙特卡罗法的非确定性算法，因此在介绍随机行走法之前，有必要介绍一下蒙特卡罗法的基本思想。蒙特卡罗（Monte Carlo）方法，又称为统计模拟方法，是一种通过反复随机采样来获得数值解的计算方法。早在18世纪，蒲丰投针问题（Buffon needle problem）就反映了统计模拟方法的思想。现代蒙特卡罗方法是在20世纪40年代由Stanislaw Ulam和von Neumann提出，该方法被广泛地应用到数学和物理领域[13]。运用蒙特卡罗法计算定积分便是其中一个重要的应用。假设存在定积分

$$\int_a^b f(x)p(x)dx \tag{2.2.1}$$

其中规定$p(x)$在区间[a,b]内是非负函数，同时满足

$$\int_a^b p(x)dx = 1 \tag{2.2.2}$$

为求解式(2.2.1)，可以利用微积分基本定理和积分方法（如分部积分法、换元法）求出解析解。当不容易求出解析解时，可以利用梯形法、矩形公式等进行数值计算，这类数值计算方法计算量较大，尤其是随着积分重数增加和定义域的扩大，计算量将会迅速增加。而蒙特卡洛积分法试图利用统计学方法来求解(2.2.1)式。我们可以将X看成一个连续型随机变量，$p(x)$是随机变量X的概率密度函数，$f(x)$是随机变量X的函数，那么根据概率论的知识可知，(2.2.1)式实际上就代表随机变量X的函数$f(x)$的数学期望。因此，求定积分的问题就转化成一个求随机变量的数学期望的问题。可以将(2.2.1)式表示成极限的形式

$$\lim_{\Delta x\to 0}\sum_{\xi=0}^{(b-a)/\Delta x} f(a+\xi\Delta x).p(a+\xi\Delta x)\Delta x \tag{2.2.3}$$

$p(a+\xi\Delta x)\Delta x$代表了随机变量X落在区间[a+$\xi\Delta x$, a+(ξ+1)Δx]的概率，可以表示成落在该区间内的样本数N_ξ与总样本数N_{Total}之比 N_ξ/N_{Total}，代入(2.2.3)式得

$$\lim_{\Delta x\to 0}\frac{1}{N_{Total}}\sum_{\xi=0}^{(b-a)/\Delta x} f(a+\xi\Delta x).N_\xi \tag{2.2.4}$$

因此，根据(2.2.4)式可以得出蒙特卡罗积分方法的一般步骤：产生一个服从$p(x)$的样本集合$\{X\}$，对该集合的每一个变量x_i，计算$f(x_i)$的值，最后计算$f(x_i)$的均值即式(2.2.1)的数值计算结果。

蒙特卡罗积分方法的优点是方法简单，不需要构建复杂的参数矩阵，计算量小，内存占用低，而且采样的过程可以并行执行，因而可以进一步加速。缺点是可能存在收敛速度慢，计算时间长的问题。

蒙特卡罗法适用于计算高斯定理的积分表达式。首先对(2.1.2)式变形，得：

$$q_i = \oiint_G \frac{1}{A_G} . A_G(-\epsilon)\nabla_r\phi(\vec{r}) . \hat{n}(\vec{r})d\vec{r} \tag{2.2.5}$$

其中A_G是高斯面的面积，可以将$1/A_G$看作高斯面上的一个均匀分布的概率密度函数，要计算q_i，可以在高斯面G上均匀采样点$\vec{r}$，计算$A_G(-\epsilon)\nabla_r\phi(\vec{r}).\hat{n}(\vec{r})$的值，最终得到的样本均值可以作为电荷量$q_i$的估计。

接下来问题的关键在于计算高斯面上的电势$\phi(\vec{r})$。随机行走算法中计算电势的关键公式是[3,6,8]

$$\phi(\vec{r}) = \oiint_{S_1} P(\vec{r},\vec{r}_1)\phi(\vec{r}_1)d\vec{r}_1 \tag{2.2.6}$$

其中$\phi(\vec{r})$表示在空间内某一点$\vec{r}$处的电势，S_1表示包围点$\vec{r}$的闭合曲面，曲面包含的空间内没有净电荷，$\vec{r}_1$代表曲面S_1上的点，$\phi(\vec{r}_1)$代表曲面S_1上点的电势，$P(\vec{r},\vec{r}_1)$表示选定一个点$\vec{r}$，从包罗面S_1上选择一个点$\vec{r}_1$的概率密度函数，又称为表面格林函数（Surface Green's Function）。该函数在整个定义域为非负值，同时

$$\oiint_{S_1} P(\vec{r},\vec{r}_1)d\vec{r}_1 = 1$$

因此，可以通过蒙特卡罗法求(2.2.5)式的数值解。根据$P(\vec{r},\vec{r}_1)$所定义的概率分布在曲面S_1上进行抽样，抽取到的$\phi(\vec{r}_1)$的均值可以作为对$\phi(\vec{r})$的估计。如果设定曲面S_1是立方体的表面，而$\vec{r}$是该立方体的中心，那么格林函数的值仅取决于$\vec{r}_1$和$\vec{r}$的相对位置以及曲面S_1所围的立方体的边长，而与$\vec{r}_1$和$\vec{r}$在空间中的绝对位置无关[6]。这一性质使得可以预先表面格林函数的数据表，在随机行走算法执行之前将数据加载进来，可以节省实时计算表面格林函数的时间，加快随机行走的速度[8]。

如果在$\vec{r}_1$处的电势也是未知的，利用(2.2.6)代入自身求$\phi(\vec{r}_1)$，形成递归：

$$\phi(\vec{r}) = \oiint_{S_1} P(\vec{r},\vec{r}_1) \oiint_{S_2} P(\vec{r}_1,\vec{r}_2) \dots \oiint_{S_k} P(\vec{r}_{k-1},\vec{r}_k)\phi(\vec{r}_k)\, d\vec{r}_k \dots d\vec{r}_2 d\vec{r}_1 \tag{2.2.7}$$

因此，在计算高斯面上一点$\vec{r}$的电势的基本过程是：

（1） 以$\vec{r}$为中心，做一个闭合的、内部无净电荷的立方体表面S_1；

（2） 根据表面格林函数决定的概率分布，在S_1上选出一点$\vec{r}_1$；

（3） 若$\vec{r}_1$点的电势仍然是未知的，那么再以$\vec{r}_1$为中心做一个闭合的、内部没有净电荷的立方体表面S_2，然后根据表面格林函数的概率分布，在S_2上选出一个点$\vec{r}_2$.....不断递归地执行。

我们把从$\vec{r}_i$到$\vec{r}_{i+1}$的过程叫做一次跳转，跳转终止的条件有两种：（1）当跳转到的某个点$\vec{r}_k$刚好处在某个导体的表面，此时$\vec{r}_k$处的电势$\phi(\vec{r}_k)$已知，跳转停止，一次随机行走结束，$\phi(\vec{r}_k)$作为$\phi(\vec{r})$的抽样样本保存；（2）随机行走跳转到离整个版图的导体的距离超过一定的范围之后，此时这次随机行走最终跳转终结在导体表面的概率很小，此时认为$\phi(\vec{r}_k)$电势为 0，终止跳转。

将式(2.2.6)代入(2.2.5)可得：

$$q_i = \oiint_G \frac{1}{A_G} A_G \epsilon \cdot (-1) \cdot \nabla_r \oiint_{S_1} P(\vec{r},\vec{r}_1)\phi(\vec{r}_1) d\vec{r}_1 . \hat{n}(\vec{r}) d\vec{r} \qquad (2.2.8)$$

因为我们只关心高斯面上的电势和电场强度，所以只需要求$\vec{r}$处的梯度∇_r，所以上式可进一步变形：

$$q_i = \oiint_G \frac{1}{A_G} \oiint_{S_1} A_G \epsilon \cdot (-1) \cdot \nabla_r P(\vec{r},\vec{r}_1)\hat{n}(\vec{r})\phi(\vec{r}_1) d\vec{r}_1 \, d\vec{r} \qquad (2.2.9)$$

对内部的二重积分项进行变形，可以利用表面格林函数构造出概率密度函数从而可以应用蒙特卡罗法进行抽样[8]：

$$q_i = \oiint_G \frac{1}{A_G} \oiint_{S_1} P(\vec{r},\vec{r}_1) A_G \epsilon \frac{-\nabla_r P(\vec{r},\vec{r}_1)}{P(\vec{r},\vec{r}_1)} \hat{n}(\vec{r})\phi(\vec{r}_1) d\vec{r}_1 \, d\vec{r} \qquad (2.2.10)$$

$$令\ \ w(\vec{r},\vec{r}_1) = A_G \epsilon \frac{-\nabla_r P(\vec{r},\vec{r}_1)}{P(\vec{r},\vec{r}_1)} \hat{n}(\vec{r}) \qquad (2.2.11)$$

称为权重函数。需要指出的是，从导体i的高斯面出发的随机行走终止在导体$j(i \neq j)$时，得到的权重值w_{ij}值是导体i和导体j之间电容C_{ij}的一次抽样；从导体i的高斯面出发的随机行走终止在导体i表面时，得到的权重值w_{ii}值是导体i的自容C_{ii}的一次抽样。因此通过随机行走可以得到一个电容矩阵

$(C_{i1} \quad C_{i2} \quad \dots \quad C_{in})$，n 为版图中导体的总数

下面讨论随机行走的终止条件。假设从导体 i 的高斯面开始进行了N_{walk}次随机行走，每次随机行走得到的权重值是w_{ik}根据[7]，导体 i 上的电荷量q_i的均值为：

$$\overline{q_i} = \frac{1}{N_{walk}} \sum_{k=1}^{N_{walk}} w_{ik} \qquad (2.2.12)$$

而q_i的方差为

$$Var_{qi} = \frac{1}{N_{walk}} \sum_{k=1}^{N_{walk}} w_{ik}^2 - \overline{q_i}^2 \qquad (2.2.13)$$

[7]中指出根据中心极限定律可知，$\overline{q_i}$服从高斯分布，而$\overline{q_i}$的标准差表示为

(2.2.14)

$$stdvar_{\overline{q_i}} = \sqrt{\frac{Var_{qi}}{N_{walk}}}$$

同理，计算电容矩阵中每个元素C_{ij}的样本均值，这就是对导体i和导体$j(j \neq i)$之间的电容值以及导体i的自容值的估计。需要指出的是，C_{ij}的样本均值指的是：

$$\overline{C_{ij}} = \frac{1}{N_{walk}} \sum_{k=1}^{n_{ij}} C_{ij\_k} \ , \tag{2.2.15}$$

n_{ij}指随机行走从导体 i 高斯面开始，到导体 j 表面停止的次数

根据[7,8]可知C_{ij}的样本均值$\overline{C_{ij}}$也近似服从高斯分布

$$Var_{C_{ij}} = \frac{1}{N_{walk}} \sum_{k=1}^{n_{ij}} C_{ij\_k}^2 - \overline{C_{ij}}^2 \tag{2.2.16}$$

C_{ij}的样本均值$\overline{C_{ij}}$的标准差

$$stdvar_{\overline{C_{ij}}} = \sqrt{\frac{Var_{C_{ij}}}{N_{walk}}} \tag{2.2.17}$$

因为$\overline{C_{ij}}$服从高斯分布，所以导体 i 和导体 j 之间的电容C_{ij}的真实值$\widetilde{C_{ij}}$有 99.7%的可能处在区间$[\overline{C_{ij}} - 3stdvar_{\overline{C_{ij}}} \ , \ \overline{C_{ij}} + 3stdvar_{\overline{C_{ij}}}]$内。因此，我们可以定义

$$err_{\overline{C_{ij}}} = \frac{3stdvar_{\overline{C_{ij}}}}{\overline{C_{ij}}} \tag{2.2.18}$$

作为$\overline{C_{ij}}$的误差。$\overline{C_{ij}}$的误差在$\pm err_{\overline{C_{ij}}}$范围内。例如可以设定当$err_{\overline{C_{ij}}}$<1%时电容值收敛，终止随机行走过程。同理，也可以计算导体 i 的电荷量q_i的样本均值$\overline{q_i}$的误差，作为判断随机行走过程是否终止的条件。

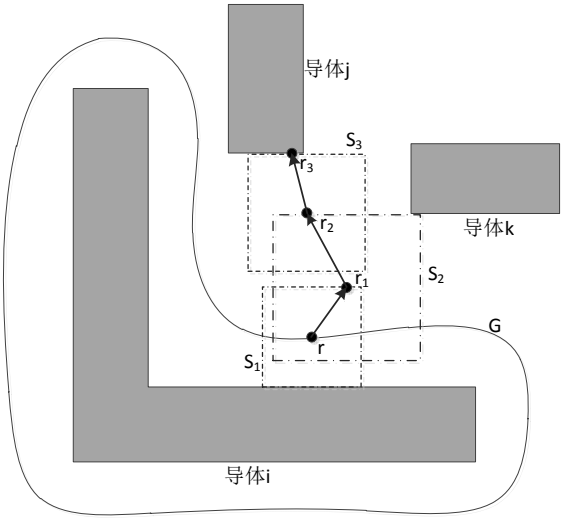


图 2.1 随机行走算法平面示意图[5]

图 2.1 给出了随机行走算法的平面示意图，根据图 2.1，可以给出随机行走算法的流程图。

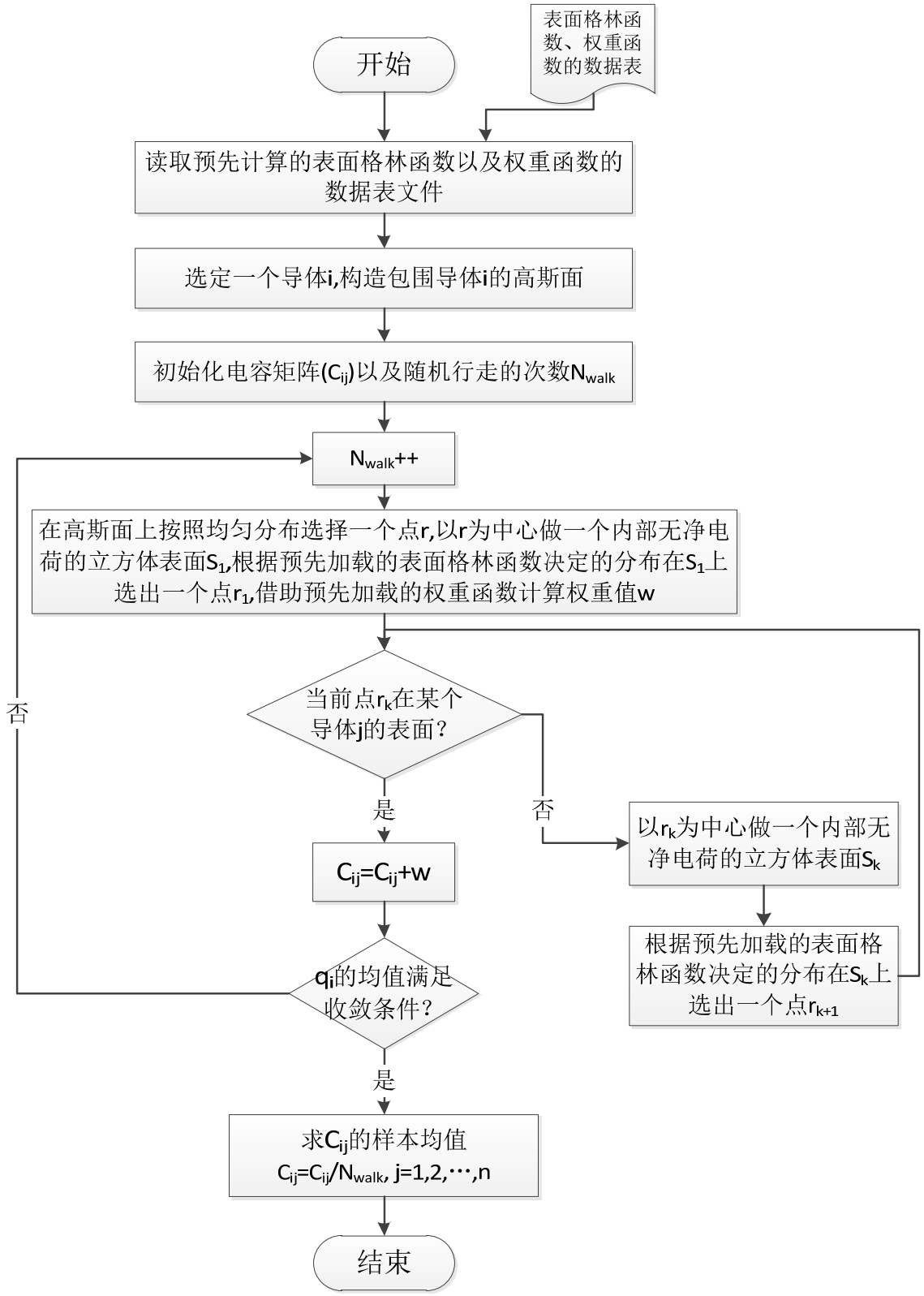


图 2.2 随机行走核心算法的基本流程

§2.3 空间管理技术介绍[9]

通过 2.2 节的分析可知，通过随机行走法进行电容提取时要进行多次随机行走，每一次随机行走又包括多次跳转，所以不考虑生成高斯面、初始化等前期准备工作的时间，随机行走的时间约为

$$T_{FRW} = N_{walk} \times N_{hop} \times T_{hop} \tag{2.3.1}$$

其中N_{walk}指的是随机行走的次数，N_{hop}指的是平均每次随机行走所包含的跳转次数，T_{hop}是平均每次跳转所需要的时间。若要加快随机行走提取电容的速度，可以从三个角度进行优化：(1)减少随机行走的次数；(2)减少一次随机行走中的跳转次数；(3)减少一次跳转所需要的时间。在本论文中重点讨论后两种优化方向。

若要一次随机行走跳转的次数尽可能少，意味着要使随机行走每次跳转幅度尽可能大，同时要尽可能增大随机行走落在导体表面的概率，这就意味着每一次跳转时产生的内部无净电荷的立方体要尽可能大，这里称之为最大转移立方体。如图 2.1 所示，若以$\vec{r}$点为中心做一个最大转移立方体，一个理论上可行的算法是：计算点$\vec{r}$到版图中各个导体表面的距离，从而得到点$\vec{r}$到版图中导体表面的距离的最小值d_{min}，该距离值实际上就对应了以$\vec{r}$点为中心的最大转移立方体的棱长的 $1/2$。按照这个算法可以确保每次跳转都可以产生最大转移立方体，然而由于每次跳转时都要计算中心点$\vec{r}_i$到版图中全部导体的距离，每次跳转的时间将会很长，平均每次随机行走的时间$N_{hop} \times T_{hop}$仍然得不到优化。为了解决上述问题，就需要对版图中导体的空间进行有效管理。

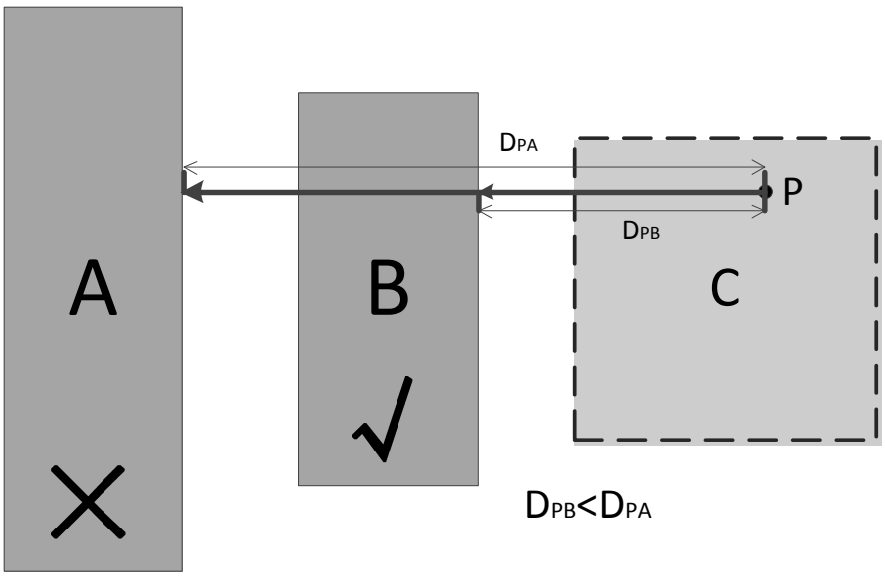


图 2.3 版图中导体空间分布特点平面示意图

如图 2.3 所示，A、B 为两个导体，C 表示空间内一个没有净电荷的区域（互连介质区）。通过观察可以发现，对于区域 C 内的任一点P，导体 B 到P的距离D_{PB}总是小于导体 A 到P的距离D_{PA}。因此以 P 为中心做最大转移立方体，那么在计算P到导体表面的最小距离时根本不需要考虑导体 A。这就给了我们一个启发：在进行随机行走之前，把整个空间细分成多个长方体空间单元 T；通过计算空间关系为每个长方体空间单元确定一组候选导体列表(CandidateList)[8]，空间单元内的某个点生成最大转移立方体时，只需要考虑该空间单元所对应的候选导体列表中的导体。采用空间管理技术之后，随机行走一次跳转的过程如图 2.4 所示。

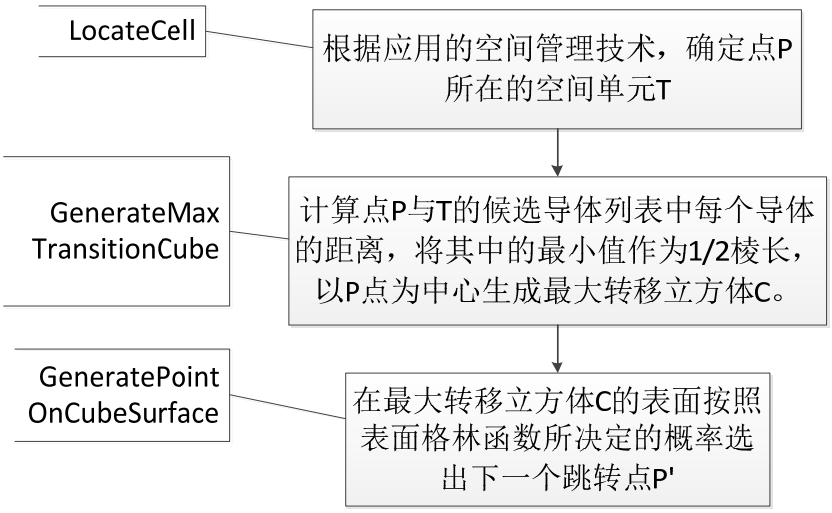


图 2.4 采用空间管理技术后的随机行走跳转过程流程图

实现上述对版图空间的管理主要涉及两个问题：(1)寻找合适的空间分割技术；(2)通过分析空间关系确定每个长方体空间单元的候选导体列表。[5]中讨论了多种空间分割技术，如维诺图 (Voronoi Diagram)、超平面 (hyperplane)、切块法(slab method)以及八叉树结构(octree)等，并分析了各方法的算法复杂度。其中，基于八叉树结构的空间分割方法可以充分利用版图中导体形状和空间位置的特点，是一种高效的空间分割方法。

第三章 空间管理技术的实现与改进

§3.1 互连线导体的分割与整合

§3.1.1 GDS 版图文件读取

GDS 版图读取程序是根据[11]提供的开源项目修改而来。GDS 文件中包含两种重要的数据结构： PATH 和 BOUNDARY。PATH 中存放了互连线导体中轴线的起始点坐标与拐点坐标，BOUNDARY 中存放了多边形导体的拐点坐标。版图处理程序读取 GDS 格式文件中的数据，对数据进行初步处理。

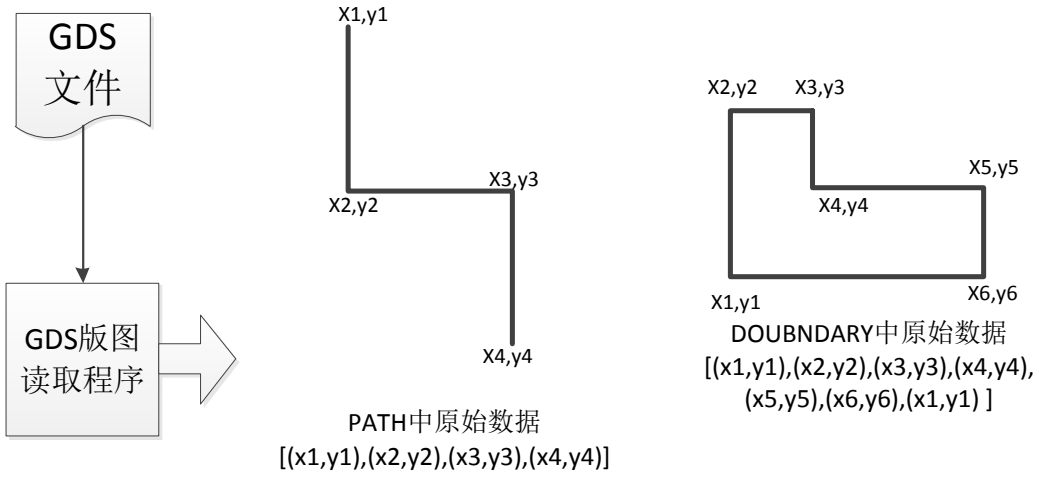


图 3.1 GDS 版图读取程序说明

§3.1.2 导体的分割

对于版图中 PATH 的分割参考了[11]的开源项目。基本的分割思路如图 3.2 所示。

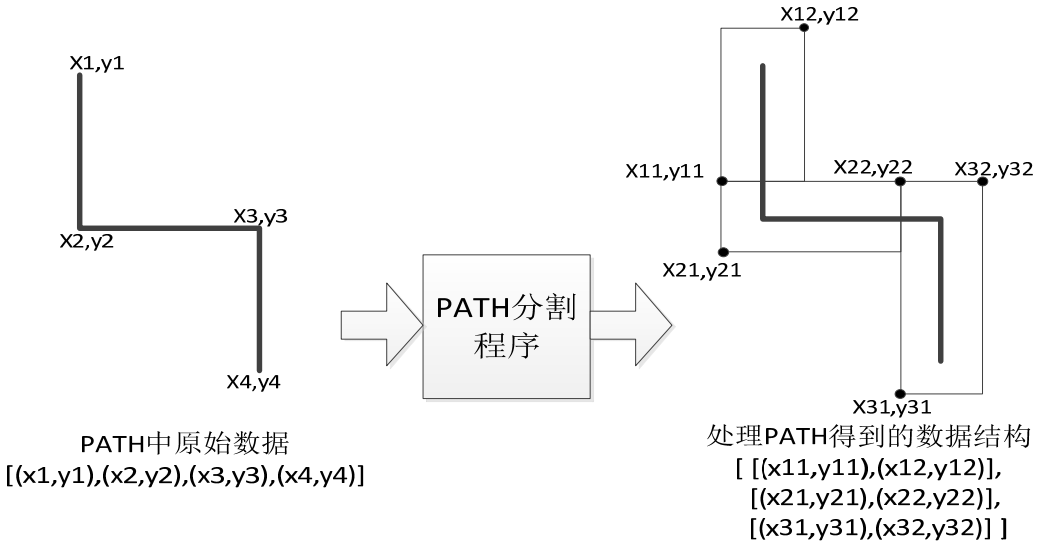


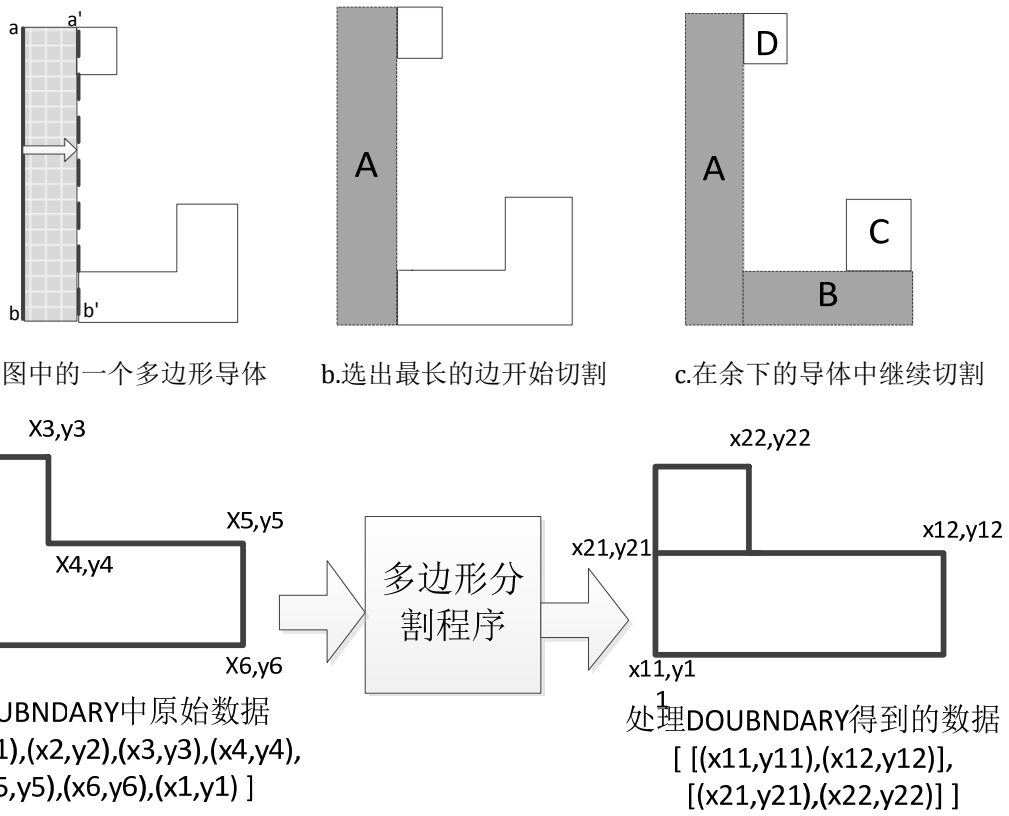
图 3.2 PATH 分割的示意图

将版图中多边形 BOUNDARY 分割成长方形的过程如图 3.1 所示。首先，读取版图信息，从中获得多边形导体各个拐点处的坐标，生成一个导体列表[11,12]；针对导体列表中的每一个导体做如下操作：

（1） 如图 3.1a 所示，从多边形中选出最长的一条边 ab，将该边向导体的内部平行移动，直至这条边碰到该导体的其它边为止，到达 a'b'处，将这条边扫过的区域切割下来，作为从多边形导体上切割的一个长方形；

（2） 在切割剩下的导体中寻找多边形，如果有多边形的话重复执行（1），直至将该导体完全切割成一组长方体。

每个长方体可以由其体对角线上两个顶点的坐标唯一确定。为了表述方便，本论文中将分割后的每个小长方体称为子导体，其所在的分割前的导体称为父导体。



a.版图中的一个多边形导体　　b.选出最长的边开始切割　　c.在余下的导体中继续切割

图 3.3 多边形导体切割成长方形示意图

每个子导体需要携带其父导体的身份信息，给出一个子导体，可以根据其携带的父导体的身份信息确定其属于哪个父导体。为实现该功能，可以首先为每个父导体添加一个唯一的身份标识符（如序列号或者随机码，身份标识符一旦确定后不可修改）；在进行导体分割生成子导体时，将父导体的身份标识符传递给每个子导体。

§3.1.2 不同互连层导体的整合

如果在版图中上下两层的互连线被中间的一个通孔(Via/Contact，本论文中不加以区分)连接，那么上下两层的互连线以及中间的通孔实际上属于同一个导体，因此需要把这三部分导体整合成一个导体。

假设版图的第 n 层是通孔层，第 n+1 层是互连线层，若要判断第 n 层的通孔Via_i和第 n+1 层的导体$Cond_j$是否相连，只需要判断是否存在$Cond_j$与Via_i在水平面上的投影是否相交即可。

如图 3.3 所示 Via 和 Conductor 分别代表通孔和导体在平面上的投影，二者相交的条件是

$$x_{c0} < x_{v1} \ \&\& \ x_{c1} > x_{v0} \ \&\& \ y_{c0} < y_{v1} \ \&\& \ y_{c1} > y_{v0} \qquad (3.1.1)$$

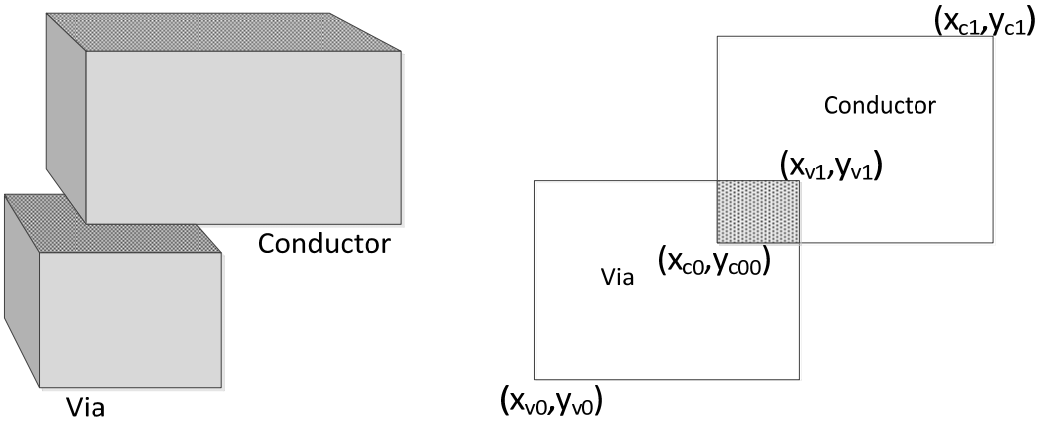


图 3.5 两个矩形交叠示意图

下面给出导体整合的一般过程：

如果通孔 V 同时与上层的导体C_t和下层的导体C_d相连，那么：

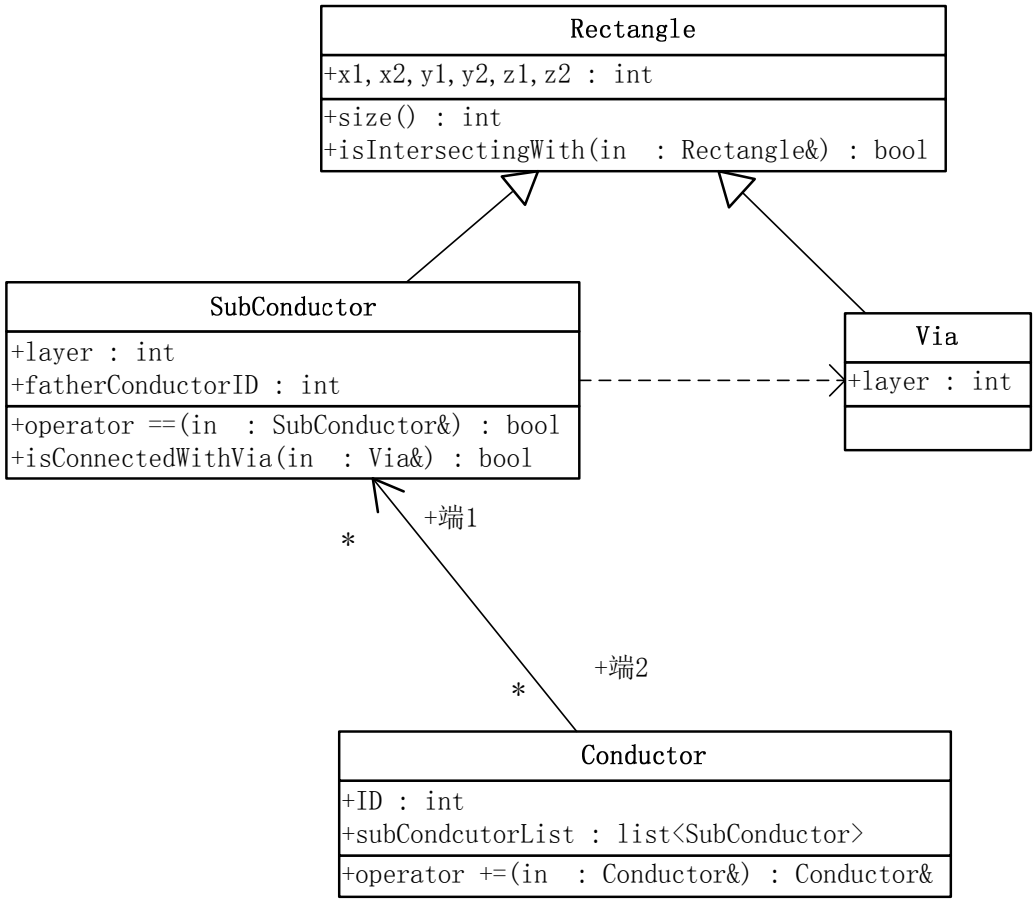
（1） 将下层导体C_d的身份标识符传递给 V，然后将V添加到C_d的子导体列表中；

（2） 将上层导体C_t的所有子导体的身份标识符修改为C_d的身份标识符，

（3） 将C_t的所有子导体并全部合并到C_d的子导体列表中，将C_t从导体列表中删除。

§3.1.3 编程实现

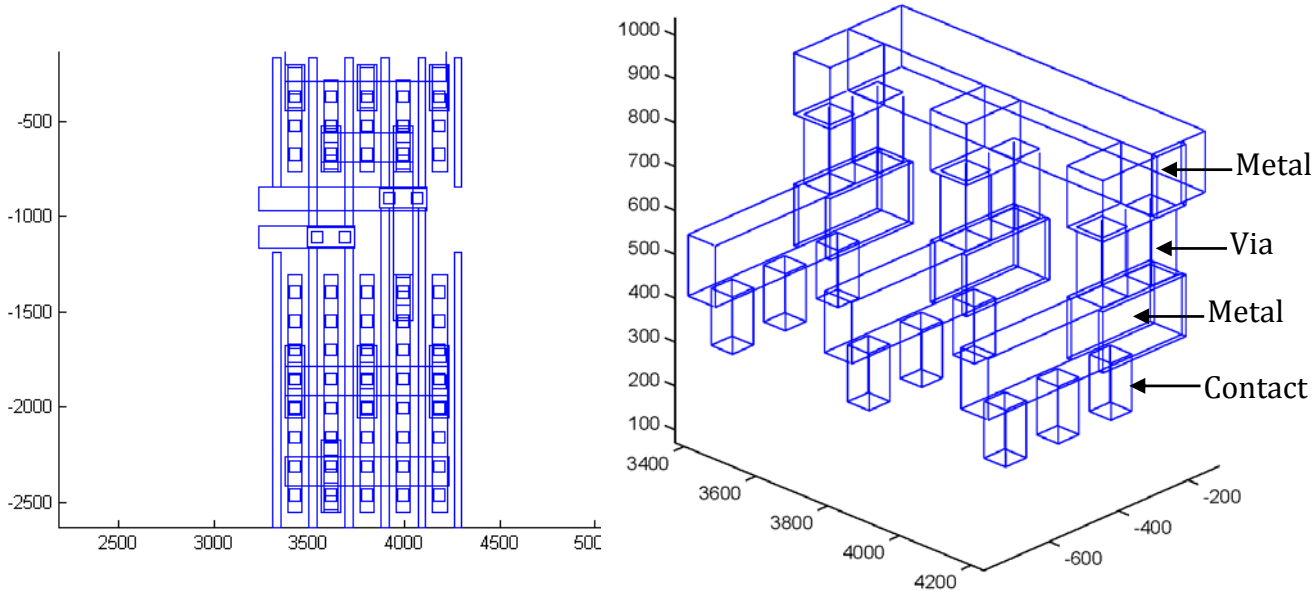
与导体的分割与整合相关的类主要有四个，分别是 Rectangle（长方体类）、SubConductor (子导体类)、Via (通孔类)、Condutcor (父导体类)。下面给出这四个类的类图以及成员说明。



Rectangle 长方体类	
int x1, x2, y1, y2, z1, z2	长方体体对角线上顶点坐标(x1, y1, z1)和(x2, y2, z2)
int size ()	返回长方体的尺寸 max(x2-x1, y2-y1, z2-1)
bool isIntersectingWith (Rectangle&)	判断两个长方体是否存在交叠
Via 通孔类：继承 Rectangle 类	
int layer	通孔在版图中的层号
SubConductor 子导体类：继承 Rectangle 类	
int layer	子导体在版图中的层号
int fatherConductorID	该子导体所属于的父导体的身份标识符
bool operator == (SubConductor &)	判断两个子导体是否是同一个子导体：若坐标完全相同，fatherConductor 相同，返回 true；否则返回 false

bool isConnectedWithVia (Via &)	判断子导体是否与通孔相接触：若通孔与子导体的层号相差 1，同时在水平面的投影有交叠，则返回 true；否则返回 false
Conductor 导体类	
int ID	导体的身份标识符，一个导体有一个唯一确定的身份标识符，并且一旦确定不可更改
list<SubConductor> subConductorList	子导体列表
Conductor& operator +=(Conductor&)	用于当两个导体被通孔连接时进行导体合并的操作符

为了直观观察导体分割和整合的效果，这里给出一个实例。首先读取与非门的版图信息，利用前述分割与整合的算法对上述二维数组进行处理，得到一个经过整合的导体列表，列表内的每个导体都包含了一个子导体列表。利用 Matlab 将全部子导体绘制出来，可以得到图 3.5 所示的效果图。图 3.5(a)是与非门互连线的平面俯视图，可以看到所有的多边形均被分割成的基本的长方形的组合；图 3.5(b)是与非门中的一个导体的三维效果图，可以看到，上下两层的互连线被通孔连接时，三者就被整合成了一个导体。由此可以验证算法的正确性。



a.与非门互连线平面俯视图　　b.与非门中的一个导体的三维效果图

图 3.6 与非门互连导体的分割与整合效果图

§3.2 空间管理的相关概念及定义

§3.2.1 基本定义&定理[9]

定义 3.1

点$P(x,y,z)$与长方体 A 在x方向上的距离是$x_{min}(\mathrm{A})-x$和$x-x_{max}(\mathrm{A})$的较大值。在y方向和z方向的距离定义类似。

$$D_x(\mathrm{P},\mathrm{A})=\max\{x_{min}(\mathrm{A})-x,x-x_{max}(\mathrm{A})\} \tag{3.2.1}$$

定义 3.2

点$P(x,y,z)$与长方体 A 的距离是P与 A 在x方向、y方向和z方向上距离的最大值。

$$\mathrm{D}(\mathrm{P},\mathrm{A})=\max\{D_x(\mathrm{P},\mathrm{A}),D_y(\mathrm{P},\mathrm{A}),D_z(\mathrm{P},\mathrm{A})\} \tag{3.2.2}$$

定义 3.3

长方体 A 与长方体 B 在x方向上的距离是$x_{min}(\mathrm{A})-x_{max}(\mathrm{B})$和$x_{min}(\mathrm{B})-x_{max}(\mathrm{A})$的较大值。在$y$方向和$z$方向的距离定义类似。

$$D_x(\mathrm{A},\mathrm{B})=\max\{x_{min}(\mathrm{A})-x_{max}(\mathrm{B}),x_{min}(\mathrm{B})-x_{max}(\mathrm{A})\} \tag{3.2.3}$$

定义 3.4

长方体 A 与长方体 B 的距离是 A 与 B 在x方向、y方向和z方向上距离的最大值。

$$\mathrm{D}(\mathrm{A},\mathrm{B})=\max\{D_x(\mathrm{A},\mathrm{B}),D_y(\mathrm{A},\mathrm{B}),D_z(\mathrm{A},\mathrm{B})\} \tag{3.2.4}$$

定理 3.1

长方体 A 与长方体 B 的距离是长方体 A 上各点到长方体 B 距离的最小值，同时也是长方体 B 上各点到长方体 A 距离的最小值。

$$\mathrm{D}(\mathrm{A},\mathrm{B})=\min\{D(P_i\in A,B)\}=\min\{D(P_i\in B,A)\} \tag{3.2.5}$$

定理 3.2

长方体 A 中的任意一点P_i到长方体 B 的最大距离等于长方体 A 的八个顶点到长方体 B 的距离的最大值。

$$\max\{D(P_i\in A,B)\}=\max\{D(P_{vertex},B)\}\ ,P_{vertex}\in\mathrm{A}\text{ 的 8 个顶点} \tag{3.2.6}$$

定义 3.5

长方体 A 的尺寸规定为长、宽、高的最大值。

$$\mathrm{Size}(\mathrm{A})=\max\{x_{max}(\mathrm{A})-x_{min}(\mathrm{A}),y_{max}(\mathrm{A})-y_{min}(\mathrm{A}),z_{max}(\mathrm{A})-z_{min}(\mathrm{A})\ \} \tag{3.2.7}$$

定义 3.6 [9]

对于长方体 A, B 和长方体空间单元 T，若 T 中任意一点P_i到长方体 A 的距离都小于P_i到长方体 B 的距离，那么长方体 A 关于 T 主导(dominate)长方体 B。

$$\text{if }\forall P_i\in T,D(P_i,A)<\mathrm{D}(P_i,B),\text{then A 关于 T 主导 B。}$$

该定义与候选导体列表的生成有密切的关系，通过后面章节的分析可以知道，长方体 T 实际上对应了空间分割后的一个空间单元；而长方体 A 或 B 则代表了子导体。空间单元 T 与长方体 A 可能的关系如图 3.6 所示：

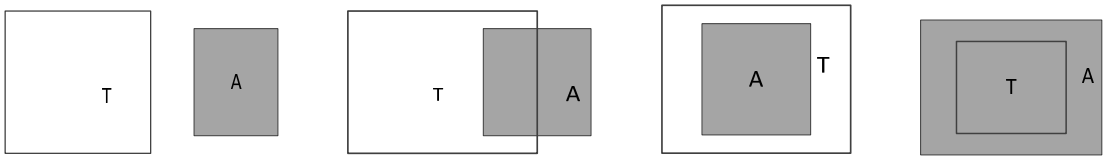
a. T 与 A 不相接触； b. T 与 A 部分相交；

c. A 被包含在 T 中； d. T 完全处于 A 的内部。

因为在随机行走过程中随机行走不可能走到某个导体的内部，所以 d 没有研究的必要。本论文中所涉及的空间单元 T 与导体 A 的关系只涉及前 3 种。因此我们可以得到一个推论：

推论 3.1

在本论文研究的问题范畴中，空间单元 T 内的点P_i到某个子导体 A 的距离的最大值$\max\{D(P_i \in T, A)\} \geq 0$。



a b c d

图 3.7 长方体 T 与长方体 A 的关系图

定理 3.3

对于长方体 A, B 和长方体空间单元 T，若 T 中的任意一点到长方体 A 最大距离小于 T 中任意一点到长方体 B 的最小距离，那么长方体 A 关于 T 主导长方体 B。

定理 3.4

由定理 1,2,3 可知，对于长方体 A, B 和长方体空间单元 T，若 T 的八个顶点到长方体 A 的最大值小于 T 与 B 的距离，那么长方体 A 关于 T 主导长方体 B。

定理 3.5

对于长方体 A, B 和长方体空间单元 T，若长方体 B 与 T 相交，那么长方体 A 关于 T 一定不能主导长方体 B。

分析：若长方体 B 与 T 相交，那么根据定理 1，$\min\{D(P_i \in T, B)\} = \mathrm{D(T, B)} < 0$; 而$\max\{D(P_i \in T, A)\} \geq 0$, 所以$\max\{D(P_i \in T, A)\} > \min\{D(P_i \in T, B)\}$，不满足定理 3 所描述的条件，因此 A 关于 T 不能主导 B。

§3.2.2 候选导体列表

为空间单元 T 生成候选导体列表的基本方法是：对于版图内的所有的子导体，检查是否要添加进空间单元 T 的候选导体列表。

下面讨论候选导体列表检查的具体方法。假设长方体空间单元为 T，存在两个长方体子导体 A, B，根据 3.2.1 节定理 3.6，如果子导体 A 关于空间单元 T 主导子导体 B，空间单元 T 中任意一点到子导体 A 的距离的最大值小于 T 中任意一点到子导体 B 的距离的最小值。这就意味着，以空间单元 T 内任意一点P_i为中心做最大转移立方体，所得到的立方体表面不可能与子导体 B 接触。因此可以得到一个重要的定理：

定理 3.6

若子导体 A 关于空间单元 T 主导子导体 B，那么 B 一定不在空间单元 T 的候选导体列表中。

为了方便判断上述主导关系，首先给出一个定义：

定义 3.7

空间单元 T 的候选导体列表中的子导体到 T 的距离的最小值与空间单元 T 的尺寸之和称为 T 的最大跳转距离(Distance Limit)。

$$L(T) = \min\{D(SC_i, T), SC_i \in CandidateList_T\} + Size(T) \qquad (3.2.8)$$

空间单元 T 的转移边界决定了以 T 中任意一点为中心跳转距离的上限，因此本文中称之为最大跳转距离。根据定义 7 可以给出两个定理：

定理 3.7

如果子导体 B 到空间单元 T 的距离 D(T, B)大于空间单元 T 的最大跳转距离$L(T)$，那么 B 一定不需要添加到 T 的候选导体列表中[9]。

证明：根据最大跳转距离$L(T)$的定义，在 T 的子导体列表中，存在子导体SC_i，满足$\max\{D(P_i \in T, SC_i)\} \le L(T)$；若 D(T, B)>$L(T)$，则 D(T, B)> $\max\{D(P_i \in T, SC_i)\}$。根据定理 1 和定理 3 可知，$SC_i$关于 T 主导 B，根据定理 3.6 可知，子导体 B 一定不在空间单元 T 的候选导体列表中。

定理 3.8

将 T 的最大跳转距离$L(T)$初始值设置为∞，通过检查空间单元 T 向外膨胀d_{ext}的区域T'内的导体生成 T 的候选导体列表后，如果此时 T 的最大跳转距离$L(T) \le d_{ext}$，那么空间单元 T 的候选导体列表是完整的。

证明：根据$L(T)$的定义，若$L(T) \le d_{ext}$，那么以空间单元 T 中任意一点为中心所做的最大转移立方体表面总会与某个子导体相接触，即每个转移立方体都是能做出的最大的立方体。因此，空间单元 T 的候选导体列表是完整的。

根据上述分析，给出检查子导体 B 是否应放进 T 的候选导体列表的算法基本流程。

CandidateCheck (SubCondcutor B, Cell T):

（1） 如果 D(B, T)> L(T)，结束；

（2） 对 T 的候选导体列表中的每一个子导体SC_i:

a) 如果SC_i关于 T 主导 B，结束；

b) 如果 B 关于 T 主导SC_i，把SC_i从 T 的候选导体列表中删除；

（3） 把 B 添加到 T 的候选导体列表中；

（4） 如果 D(B, T)+size(T)<L(T)，则令 L(T)= D(B, T)+size(T)；

（5） 结束。

根据上述算法的过程，我们可以得到一个定理：

定理 3.9

空间单元 T 的最大跳转距离$L(T)$的初始值对应了在为 T 生成不完整候选导体列表时向外膨胀的距离。

根据上述定理，如果 T 的最大跳转距离$L(T)$初始值是∞，那么在对空间单元 T 进行候选导体列表检查时实际上是对版图空间中的所有子导体进行检查。然而对检查版图内的所有子导体的方式较为耗时。

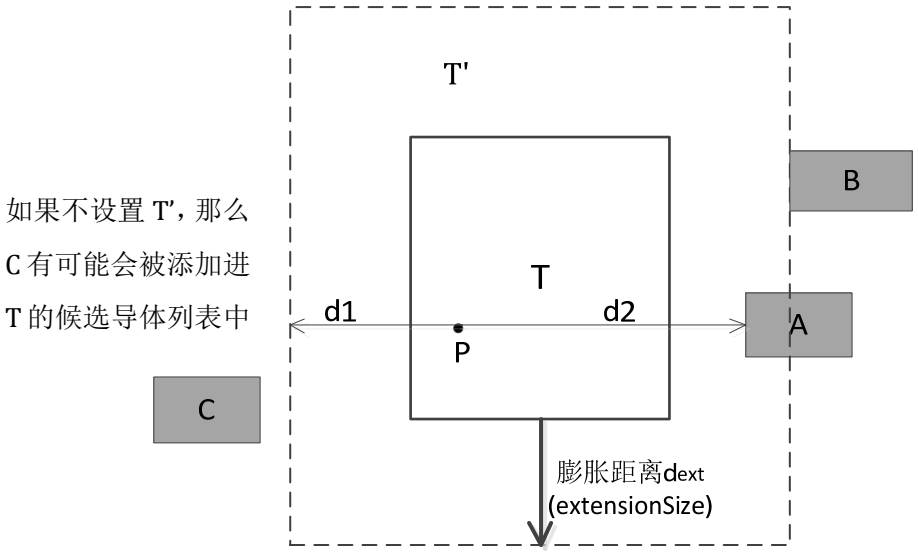


图 3.8 不完整候选导体列生成二维示意图

[9]提出一种**“不完整候选导体列表”**的方法：如图 3.7 所示，在为每个空间单元 T 进行候选导体列表检测之前，令 T 的最大跳转距离$L(T)$初始值设置为一个有限的常

数d_{ext}。这等价于在为 T 生成候选导体列表时，从 T 的表面向外膨胀一定的距离d_{ext}，得到膨胀后的空间区域T'；所有与膨胀后的空间区域T'接触或相交的子导体（如 A 和 B）需要检查是否添加进入空间单元 T 的候选导体列表，其余子导体（如 C）均不考虑。**根据定理 3.9，将 T 的最大跳转距离$L(T)$初始值d_{ext}称为 T 的膨胀距离(extensionSize)。**

如图 3.7 所示，之所以称该方法为“不完整候选导体列表”，是因为该方法设置了需要检查的子导体的初始范 T'。因为这个初始范围的存在，子导体 C 便一定不会添加到空间单元 T 的候选导体列表中；而如果没有设置初始范围，那么子导体 C 也有可能会被添加到 T 的候选导体列表中。换言之，此时空间单元 T 的候选导体列表可能是不完整的。

在随机行走过程中，如果某一次跳转到的点 P 落在空间单元 T 内，在以 P 为中心产生最大转移立方体时，首先计算点 P 与 T 的候选导体列表中子导体之间的最小距离，如图 3.7 中的d_2；然后计算点 P 到膨胀后的空间区域T'表面的距离，如图 3.7 中的d_1；比较d_1与d_2的值，将二者的较小的值作为以 P 点为中心的最大转移立方体棱长的 $1/2$。

使用“不完整候选导体列表”方法的优点：

（1） 减少为每个单元生成候选导体列表的时间，从而加快生成八叉树的过程；

（2） 减小候选导体列表的平均长度，减小随机行走时遍历候选导体列表计算点到候选子导体之间的距离所用的时间。

使用“不完整候选导体列表”方法的注意事项：

（1） 合理设置常数d_{ext}的值（即最大跳转距离 L(T)的初始值）。如果d_{ext}值过大，那么生成候选导体列表的加速效果不明显；如果d_{ext}值过小，那么会导致随机行走过程中每次跳转的“步子”减小，从而增加跳转时间。

假设某次随机行走跳转至点 P，并且已知 P 所在的空间单元是 T。下面给出上述过程的算法流程：

GenerateMaxTransitionCube (P, T):

（1） 将空间单元 T 的坐标(x_1, y_1, z_1)~(x_2, y_2, z_2)向外膨胀设定的距离d_{ext}，得到膨胀后的空间单元T'的坐标

$$(x_1 - d_{ext}, y_1 - d_{ext}, z_1 - d_{ext})\sim(x_2 + d_{ext}, y_2 + d_{ext}, z_2 + d_{ext})$$

（2） 计算点 P 到膨胀的空间单元T'的表面的距离d_1；

（3） 遍历空间单元 T 的候选导体列表，计算 P 到 T 的候选导体列表中的子导体$subCond_i$的最短距离d_2；

（4） 比较d_1和d_2，以二者中较小的值作为最大转移立方体的棱长的 $1/2$，生成最大转移立方体。

§3.3 空间划分方法

§3.3.1 三维数组法

三维数组法的基本思路是将版图的三维互连线区域划分成大小相等的三维栅格，将每个栅格作为一个基本的空间单元，然后按照一定的规则为每一个栅格生成候选导体列表。主要涉及两个问题：栅格的划分方式以及每个栅格的形状，以及生成候选导体列表时需要考虑的导体的范围。下面分别讨论这两问题。

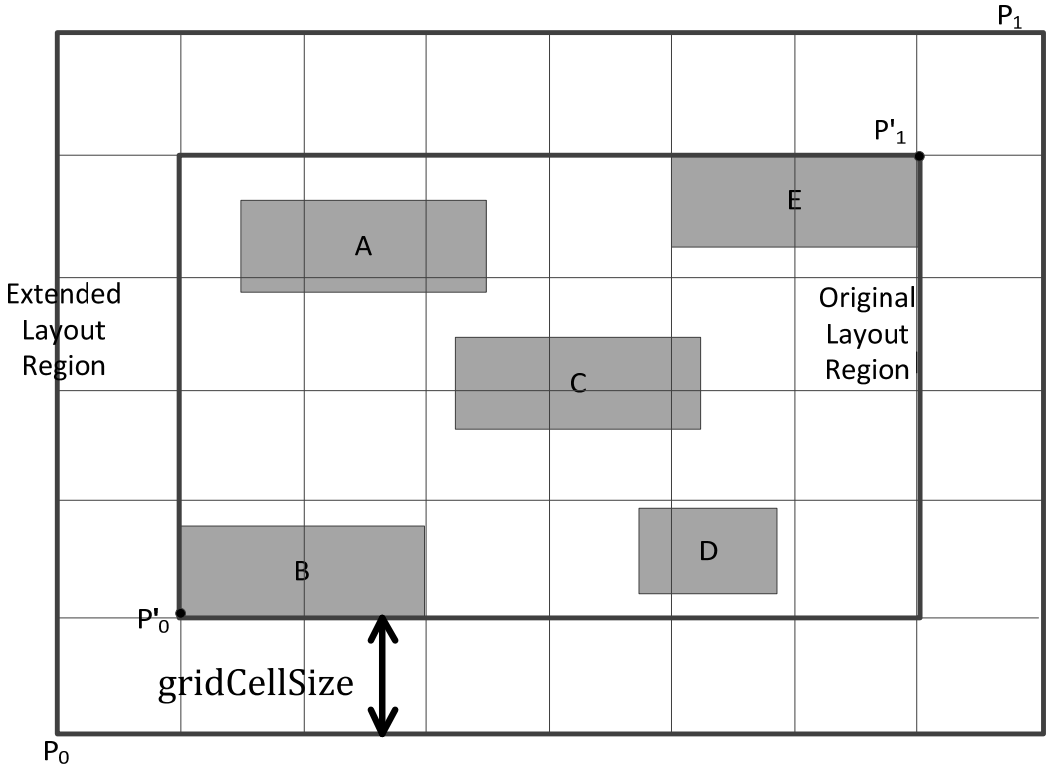


图 3.9 空间划分平面示意图

如图 3.8 所示，假设版图中有子导体 A, B, C, D, E，基本处理步骤如下：

（1） 计算得到版图中所有子导体在x, y, z方向上最小坐标P_0'和最大坐标P_1'，由P_0'和P_1'确定的区域包含了版图中所有的导体，这里称其为初始版图区域；

（2） 为了便于处理随机行走跳出版图边界的情况，需要把初始版图向外膨胀一定的距离，为方便把向外膨胀的距离设置为栅格尺寸(GridCellSize)。

（3） 以栅格尺寸为单位，将扩展后的版图区域分割成大小相等的栅格，每个栅格都是棱长为 GridCellSize 的立方体，每个栅格都作为三维数组中的一个元素存储。

根据上面的讨论，每个栅格都是一个空间单元，接下来的工作就是为每一个栅格生成候选导体列表。将每个栅格 T 的最大跳转距离 L(T)的初始值d_{ext}设置为栅格的棱长：

$$d_{ext} = gridCellSize$$

采用 3.2.2 节所介绍的不完整候选导体列表的思路为每个栅格生成不完整候选导体列表。

使用三维数组的空间分割方法的优点：

（1） 是给定某点的坐标，可以迅速定位并访问该点所在的空间单元（栅格）；

使用三维数组的空间分割方法的缺点：

（1） 如果要使每个栅格的候选导体列表的长度变短，需要减小栅格的大小，栅格的数量会迅速增加，对内存的占用较大；当栅格减小时，生成的候选导体列表的不完整率可能增加，从而使的随机行走的速度减慢，**在本实验中，将栅格尺寸规定为版图互连层厚度的 1/2:**

$$gridCellSize = 1/2 \times \text{版图互连层厚度}$$

（2） 由于三维数组是对空间进行均匀分割，在导体较稀疏的区域可能出现过度划分的情况。

§3.3.2 基于八叉树结构的空间划分

八叉树算法是一种使用树结构对空间进行划分的方法。图 3.9 给出了基于八叉树结构空间划分的平面示意图。为便于理解可以将每一个黑点理解成一个候选导体，当一个空间单元的候选导体的数目大于一定值（这里假设是 1）时，将该三维空间单元等分成 8 个子单元，然后在每个子单元中再进行上述检测，直至空间单元中的候选导体数目不大于 1 或者空间单元的尺寸达到临界值时停止。

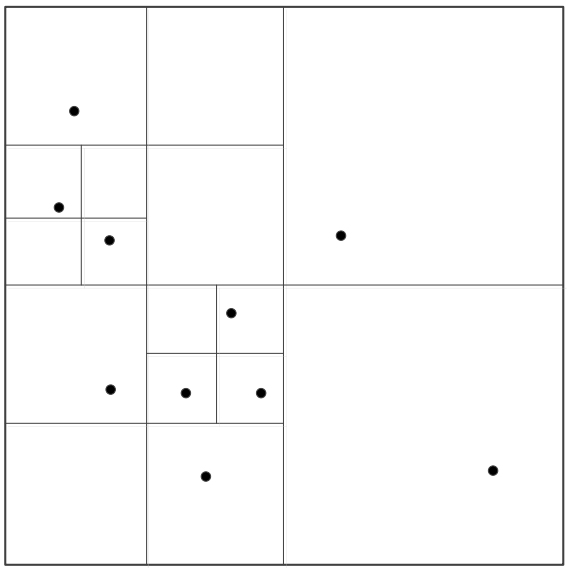


图 3.10 八叉树结构空间划分平面示意图

产生八叉树结构空间划分的函数算法流程如下：

GenerateOctree (SubConductor subCond, Cell cellT)

（1） 如果D(subCond, cellT) $>$ $L(cellT)$，那么返回，否则执行（2）；

（2） 如果 cellT 是叶节点（没有子空间单元），那么执行（3），否则执行（5）；

（3） 调用候选导体序列检查函数 CandidateCheck(subCond, cellT)，检查是否要把 subCond 添加进 cellT 的子导体列表中；

（4） 如果 **cellT 的候选导体列表的长度 $>$ 规定的上限，并且 cellT 的尺寸 $>$ 规定的尺寸下限**，那么：

a) 将 cellT 分成 8 个大小相等的子空间单元节点$subCell_{i=1,\ldots,8}$，对于每一个子空间节点$subCell_i$：

i. 依次取出 cellT 的候选导体列表中的每个子导体candidate$Cond_i$；

ii. 递归调用 GenerateOctree($candidateCond_i$ ，$subCell_i$);

（5） 对于 cellT 每一个子空间单元$subCell_{i=1,\ldots,8}$：

a) 递归调用 GenerateOctree(subCond，$subCell_i$);

用八叉树结构进行空间划分的优点：

（1） 充分利用导体的空间分布特点，降低叶节点的数目，节省内存占用。

（2） 如果在八叉树根节点的候选导体列表是完整的，那么内部的叶节点空间单元的候选导体列表的完整率会很高。

§3.3.3 组合型空间划分方法

在上述两种方法的基础之上，[9]提出一种组合型空间划分方法。如图 3.10 所示:

① 按照 3.3.1 节所介绍的三维数组的空间划分方法将版图划分成立方体栅格，并存放到三维数组中；将每个栅格的最大跳转距离 L 的初始值d_{ext}设置为栅格的尺寸(gridCellSize)，为每个栅格生成候选导体列表；

② 将每个栅格作为一个八叉树结构的根节点$rootCell_i$，对于根节点候选导体列表中的每个子导体$subCond_j$，调用 GenerateOctree($subCond_j$,$rootCell_i$) 产生八叉树结构。

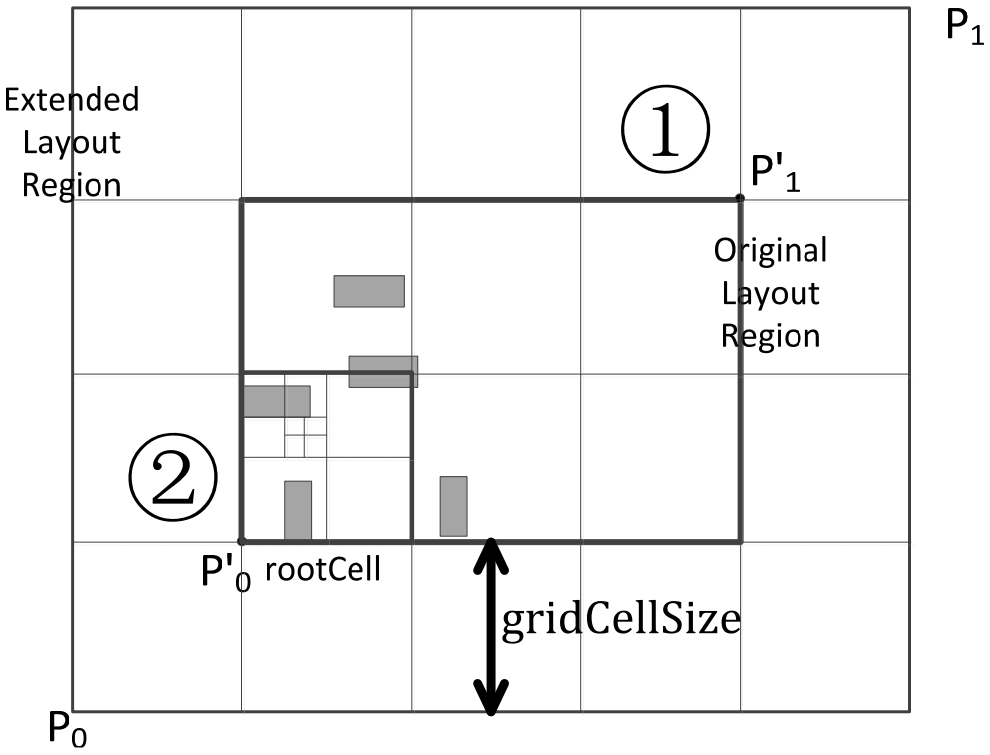


图 3.11 组合型空间划分方法平面示意图

需要说明的是，在生成八叉树时，子节点空间单元与父节点空间单元的最大跳转距离 L 的初始值d_{ext}相同，且都为栅格的尺寸:

$$d_{ext} = gridCellSize$$

下面给出组合型空间划分方法的具体的算法流程：

GenerateGridOctree:

（1） 按照 3.3.1 节所介绍的三维数组的空间划分方法将版图划分成立方体栅格，将每个栅格作为一个空间单元存放进三维数组 G 中。假设三维数组 G 包含$N_X \times N_Y \times N_Z$个空间单元，整个版图空间的最小坐标为$(x_0, y_0, z_0)$；

（2） 设 gridCellSize 等于三维数组中每个栅格的尺寸，规定空间划分的每个空间单元 T（包括三维数组的栅格、八叉树结构的根节点与叶子节点）最大转移边界L(T)的初始值设置为栅格尺寸 gridCellSize；

（3） 为三维数组中的每个栅格（空间单元）进行候选导体列表检测。

对于版图中的每一个子导体 subCond:

a) 获得 subCond 的最小坐标$(x_{min}, y_{min}, z_{min})$和最大坐标$(x_{max}, y_{max}, z_{max})$

b) nx1=max $(\lfloor (x_{min} - x_0)/gridCellSize \rfloor - 1, 0)$;

nx2=min $(\lceil (x_{max} - x_0)/gridCellSize \rceil, N_X - 1)$;

ny1=max $(\lfloor (y_{min} - y_0)/gridCellSize \rfloor - 1, 0)$;

ny2=min $(\lceil (y_{max} - y_0)/gridCellSize \rceil, N_Y - 1)$;

nz1=max $(\lfloor (z_{min} - z_0)/gridCellSize \rfloor - 1, 0)$;

nz2=min $(\lceil (z_{max} - z_0)/gridCellSize \rceil, N_Z - 1)$;

c) 对三维数组中 i 从 nx1 到 nx2，j 从 ny1 到 ny2，k 从 nz1 到 nz2 的每一个栅格单元 G[i][j][k]:

i. 调用函数 CandidateCheck(subCond, G[i][j][k]);

（4） 以三维数组中的每一个栅格单元 G[i][j][k]为根节点产生八叉树：

a) 对 G[i][j][k]的候选导体列表中的每一个子导体SC_i:

i. 调用函数 GenerateOctree(SC_i, G[i][j][k]);

下面介绍一下采用组合型空间管理技术条件下定位节点单元的基本流程。

假设整个版图空间的最小坐标是(x_0, y_0, z_0)，已经利用 3.3.3 节介绍的算法生成了三维数组 G$[N_X][N_X][N_X]$，三维数组的每个栅格的尺寸为 gridCellSize。并且以三维数组中的每个元素所对应的栅格为根节点生成了八叉树。现在空间内有一点$P(x, y, z)$，要定位点 P 所在的叶节点空间单元，基本的算法流程如下。

首先定位点 P 在三维数组 G 中的位置，获得 P 所在的八叉树的根节点。

LocateOctreeRootCell (P, G)：

（1） $x_n = \lfloor (x - x_0)/gridCellSize \rfloor$,

$y_n = \lfloor (y - y_0)/gridCellSize \rfloor$,

$z_n = \lfloor (z - z_0)/gridCellSize \rfloor$;

（2） 返回 G$[x_n][y_n][z_n]$，即点 P 所在的八叉树的根节点。

然后根据点 P 的坐标和八叉树的根节点，确定点 P 所在的叶节点空间单元。LocateCellFromRoot (P, rootCell):

（1） 若 rootCell 是叶节点，返回 rootCell；

（2） 对于 rootCell 的子节点单元列表中的每个子空间节点$subCell_{i,i=1,2,\ldots,8}$:

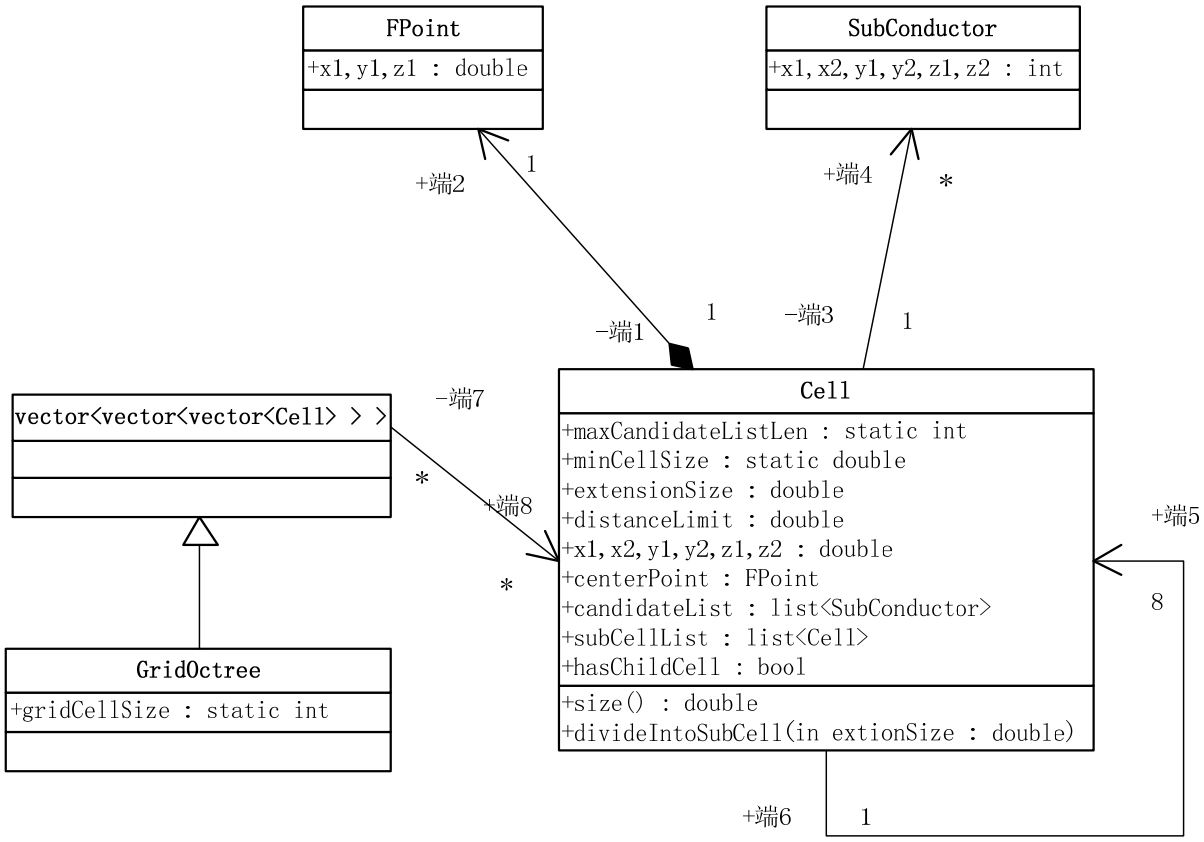
a) 比较 P 与$subCell_i$的坐标，若 P 在$subCell_i$内部或者表面：

i. 递归调用 leafCell=LocateCellFromRoot(P, $subCell_i$);

ii. 返回 leafCell;

§3.3.4 编程实现

空间管理技术编程实现时涉及到多个数据结构，如最大转移立方体、候选导体列表(CandidateList)、三维数组、八叉树结构(Octree)以及八叉树中的节点空间单元(Cell)。其中空间单元类 Cell 是整个空间管理技术关键的数据结构，下面将给出上述各个数据结构的内部成员以及相互关系。

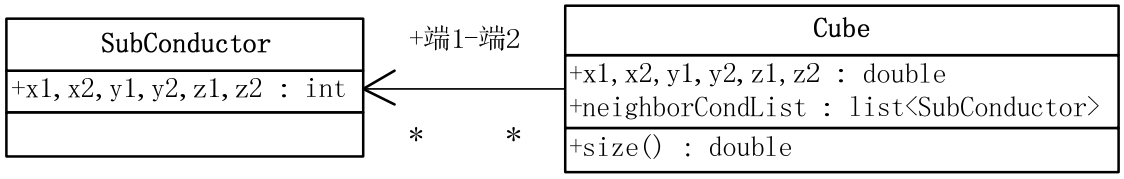


FPoint 点类	
double x1, y1, z1	点在空间中的坐标(x1, y1, z1)
SubConductor 子导体类：继承 Rectangle 类	
int x1, x2, y1, y2, z1, z2	子导体的坐标(x1, y1, z1)和(x2, y2, z2)
Cell 空间单元类 每个空间单元都是八叉树中的一个节点	
static int maxCandidateListLen	空间单元候选导体列表规定的最大长度，若某个空间单元的导体列表的长度大于该值并且该空间单元的大小大于 minCellSize，那么需要将该空间单元划分成 8 个相等的子单元。
static double minCellSize	空间单元规定的最小尺寸
double extensionSize	在生成不完整候选导体列表时空间单元向外膨胀的距离d_{ext}
double distanceLimit	空间单元 T 的最大跳转距离$L(T)$
double x1, x2, y1, y2, z1, z2	空间单元的坐标(x1, y1, z1)和(x2, y2, z2)
FPoint centerPoint	空间单元的中心点，用来存放空间单元的中心点坐标
list<SubConductor> candidateList	空间单元的候选导体列表，使用 list，list 中的每个元素都是一个子导体 SubConductor 对象
list<Cell> subCellList	空间单元的子节点列表，这个成员是构造八叉树结构的关键：当一个空间单元需要分成 8 个相等的子单元时，需要创建 8 个 Cell 对象，设定好其坐标，然后将这个 8 个 Cell 对象放进 subCellList 中。
bool hasChildCell	判断当前空间单元是否有子节点（子空间单元）。如果当前空间单元的候选导体列表的长度为 0，那么 hasChildCell=false，表示当前空间单元是一个叶子节点；否则，hasChildCell=true，表示当前空间单元不是一个叶子节点。
double size()	返回当前空间单元的尺寸
divideIntoSubCell (double extensionSize)	将当前空间单元分成 8 个子单元。 假设 centerPoint 中的坐标为(c_x, c_y, c_z)，当前空间单元的坐标为(x1, y1, z1)和(x2, y2, z2)，那么划分的 8 个子单元的坐标分别为： (x1, y1, z1)~(c_x, c_y, c_z) (c_x, y1, z1)~(x2, c_y, c_z) (x1, c_y, z1)~(c_x, y2, c_z) (c_x, c_y, z1)~(x2, y2, c_z) (x1, y1, c_z)~(c_x, c_y, z2) (c_x, y1, c_z)~(x2, c_y, z2)

	(x1, c_y, c_z)~(c_x, y2, z2) (c_x, c_y, c_z)~(x2, y2, z2) 参数 extensionSize 的作用是用来设置子单元的膨胀距离。
GridOctree 组合型八叉树类，继承自三维数组 vector<vector<vector<Cell> > > 三维数组的每一个元素都是一个八叉树的根节点。	
static int gridCellSize	三维数组中每一个栅格所对应的空间单元的大小，在组合型空间划分中一般该值相对较大，可以根据版图高度灵活设置。

图 3.11 与空间划分相关的数据结构及成员描述

在随机行走过程中每次跳转都需要产生一个最大转移立方体。下面给出最大转移立方体的数据结构。



SubConductor 子导体类：继承 Rectangle 类	
int x1, x2, y1, y2, z1, z2	子导体的坐标(x1, y1, z1)和(x2, y2, z2)
Cube 最大转移立方体类	
double x1, x2, y1, y2, z1, z2	最大转移立方体 Cube 的坐标
list<SubConductor> neighborCondList	相邻导体列表，即与最大转移立方体表面相接触的子导体列表。在以空间单元 T 内一点 P 为中心生成最大转移立方体 cube 时，将 T 的候选导体列表中到点 P 距离最短的子导体放进 neighborCondList 中。当从最大转移立方体 cube 的表面选出下一个跳转点P'时，若点P'是否在 cube 的相邻导体列表中的子导体表面上，那么表示随机行走跳转到了某个导体表面，随机行走停止；否则继续执行下一步跳转。
double size()	返回最大转移立方体的棱长

图 3.13 与最大转移立方体相关的数据结构及成员描述

§3.4 空间管理中的加速技术

§3.4.1 产生八叉树结构的加速方法

根据 3.3.3 节对组合型空间划分方法的介绍，在生成八叉树时，子节点空间单元与父节点空间单元的最大跳转距离 L 的初始值d_{ext}相同，且都为栅格的尺寸。换言之，空间单元的膨胀距离 extensionSize 是固定值。这种处理方式可能带来两个问题：

1. 在为空间单元生成不完整候选导体列表时，可能存在空间单元很小，但生成候选导体列表时检测的子导体的范围很大的情况。

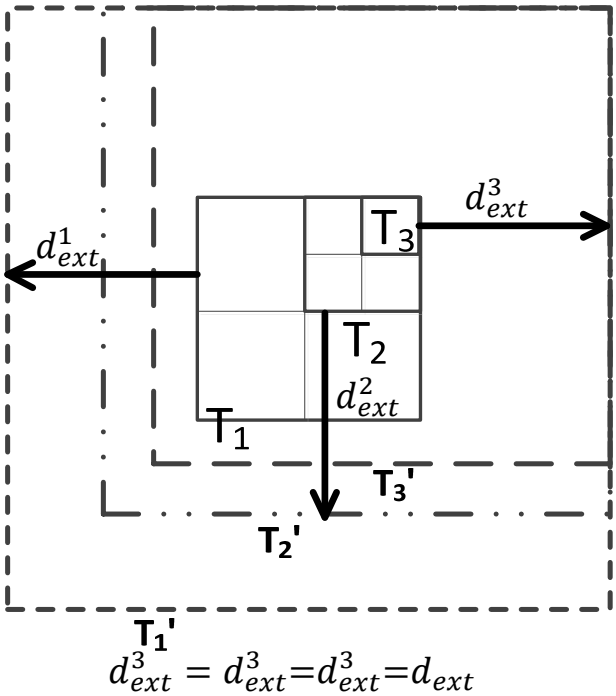


图 3.14 候选导体列表冗余检查平面示意图

如图 3.13，空间单元从 T1 分割到 T3，尺寸迅速减小，但空间单元最大跳转距离 L 的初始值始终为d_{ext}。此时 T3 的尺寸很小，但 T3 生成候选导体列表时，需要检查的子导体范围T_3'很大。而实际上在生成T_3的不完整候选导体列表时可能只需要检测其附近一个较小的区域内的子导体，而与T_3'中距离空间单元T_3较远的子导体无关，这就导致存在冗余检测的情况。

2. 八叉树存在过度分割情况。

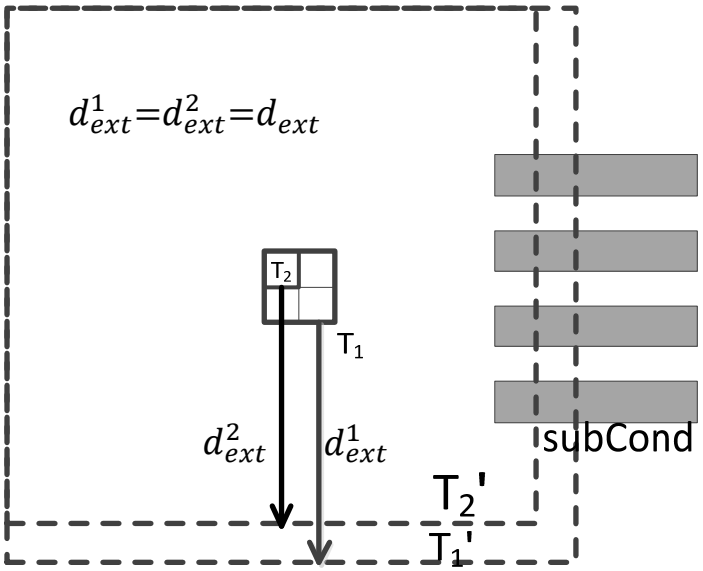


图 3.15 八叉树过度分割平面示意图

类似图 3.14 所示的情况，右侧是 4 个子导体，T1 和 T2 的最大跳转距离 L 的初始值d_{ext}^1和d_{ext}^2是一个相同的固定值。无论把空间单元 T1 进行多少次划分，都无法将右侧 4 个子导体从候选导体列表中除去。因此，这将会导致空间单元的过度分割。空间单元的过度分割会增加生成八叉树的时间，同时八叉树的平均深度会增加，从而导致随机行走过程中定位八叉树中的叶子节点的时间增加。

针对上述两个问题，本文提出一种可行的优化方法：

在生成八叉树的过程中，将子节点空间单元的最大跳转距离$L_{子节点}$的初始值$d_{ext子节点}$设置为当前父节点空间单元的最大跳转距离$L_{父节点}$乘以一个缩放因子k，即：

$$d_{ext子节点} = k \times L_{父节点} \quad , k \leq 1$$

根据定理 3.9，上述方法的基本思路是减小子节点的膨胀距离。

该改进方法的优点：

（1） 若$k < 1$，八叉树在从父节点划分成子节点时，子节点空间单元的膨胀距离也将减小，所以当空间单元较小时，其对应的膨胀距离也较小，因此可以降低冗余检测的比率。

（2） 如图 3.15 所示，当空间单元 T1,分割成 T2 时，T2 的膨胀距离d_{ext}^2小于 T1 的膨胀距离d_{ext}^1，此时右侧的 4 个子导体与T_2'不相交，根据 3.2.2 节介绍的不完整候选导体列表的生成方法，右侧 4 个子导体均不会添加进 T2 的候选导体列表中，因此避免了对空间单元的过度分割。

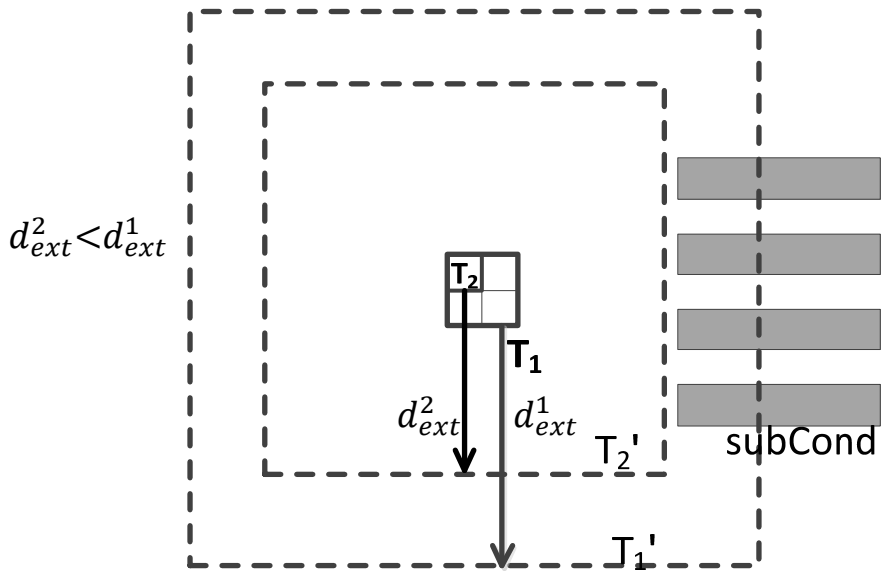


图 3.16 优化后八叉树分割平面示意图

（3） 通过选择合适的k值，可以在减少八叉树生成时间的同时，又可以保证降低随机行走的速度不下降。原因有如下几点：

a) 改进方法使八叉树的叶节点空间单元的不完整候选导体列表的平均长度下降。减小了在生成最大转移立方体时遍历空间单元的不完整候选导体列表所消耗的时间；

b) 上述优化方法改善了对空间单元过度分割的情况，八叉树的平均层数减少，因此随机行走过程中定位某一点所在的空间单元的平均时间下降。

基于上述原因，尽管该优化方法可能会导致候选导体列表的不完整率增加，从而平均每次随机行走的跳转数增加，但总体上随机行走的速度并没有受到影响。

§3.4.2 定位节点单元的加速方法

方法一

在随机行走过程中，从当前点跳转到下一个点时，首先检查下一个点是否仍在当前的空间单元里。

如图 3.16 所示，当前点 P1 所在的空间单元是 T；按照 2.2 节介绍的随机行走的过程，跳转到下一个点 P2。在确定 P2 所在的空间单元的时候，首先判断一下 P2 是否仍然在空间单元 T 中。若 P2 刚好在 T 中，那么一步就确定了 P2 所在的空间单元；若 P2 不在 T 中，再按照 3.3.3 节介绍的定位节点单元的方法进行定位。

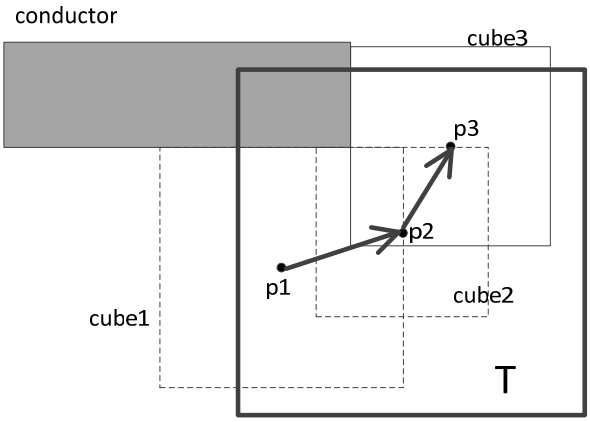


图 3.17 随机行走跳转示意图

我们称 P2 仍在 T 中的这种情况为命中（hit）。随机行走中过程中命中的次数占总次数的比重称为命中率（r）。假设判断P_2是否仍在空间单元 T 所用的时间是t_1，按照传统方法从八叉树根节点定位P_2所在的叶节点单元所用的时间是t_2，命中率是r，那么定位节点单元的总时间可以近似表示为

$$t_{locate} = r \times t_1 + (1-r) \times (t_1 + t_2) = t_1 + (1-r)t_2 \tag{3.4.1}$$

定位节点单元法带来的加速时间

$$t_{speedup} = t_2 - (t_1 + (1-r)t_2) = rt_2 - t_1 \tag{3.4.2}$$

定位节点单元法带来的加速百分比

$$P_{speedup} = \frac{t_{speedup}}{t_2} = r - \frac{t_1}{t_2} \tag{3.4.3}$$

方法二

为八叉树的节点创建索引，利用索引加快定位某一点所在叶节点单元的速度。

为了便于说明，这里以为四叉树为例进行分析。如图 3.17 左图所示，

（1） 将平面内的一个正方形通过四叉树结构进行分割，得到一个深度为 3 的四叉树。

（2） 以该四叉树第 2 层的节点单元的尺寸作为基本单位，将该正方形打成一个个大小相等的小栅格，可以得到一个 4X4 的网格，该网格左下角是第(0,0)号栅格，向右一个为第(1,0)号栅格……依此类推。

（3） 创建一个 4X4 的指针数组 G，该数组的每个元素与（2）中网格相对应。例如 G[1][2]就对应了网格中第(1,2)号栅格。本文称该数组为**“八叉树索引数组”**。

（4） 通过比较坐标确定第(i,j)号栅格在八叉树的哪个节点里，确定之后在指针数组 G[i][j]处存放该节点的指针。例如第(0,0)号栅格处在八叉树第一层最左侧的节点中，因此 G[0][0]指向八叉树第一层最左侧的节点（如图中箭头所示）。

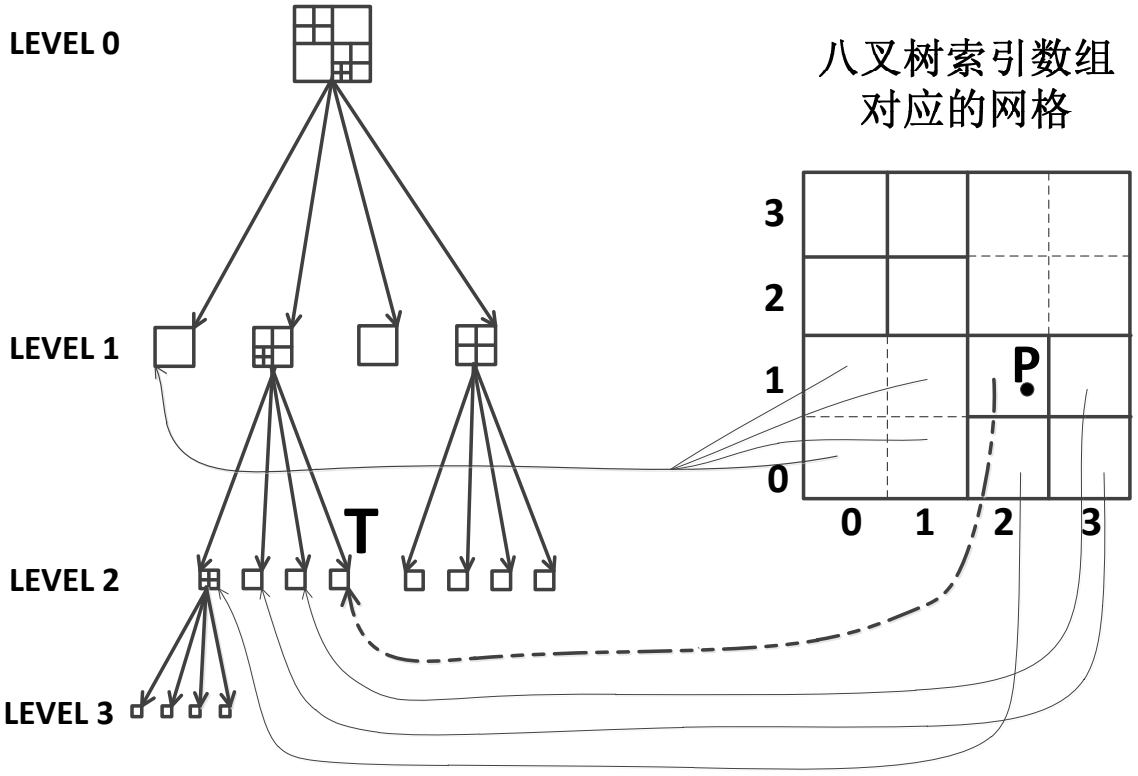


图 3.18 为八叉树建立索引平面示意图

需要注意的几点：

（1） 并不是指针数组的每个元素都指向八叉树的叶子节点，例如图 3.17 中的 G[2][0]就指向了八叉树 LEVEL2 的一个父节点；

（2） 有可能指针数组的多个元素指向八叉树的同一个节点，例如图 3.17 中的 G[0][0]、G[1][0]、G[0][1]、G[1][1]都是指向八叉树 LEVEL1 最左侧的节点；

（3） 若已知点 P 的坐标，确定点 P 所在的八叉树的叶子节点的方法时，首先计算出 P 在网格中的哪一个栅格，如图 3.17 中 P 在网格的第(2,1)号栅格；然后访问 G[2][1]指向的八叉树节点 T，从该节点开始继续向下定位 P 所在的叶节点。

根据上述分析，假设已经生成了八叉树，根节点的最小坐标为(x_0, y_0, z_0)，八叉树索引数组的栅格尺寸为S'，给出为根节点生成对应的八叉树索引数组的算法流程：

GenerateSubGridOfRootCell (Cell cellT, SubGrid G_S):

（1） 如果 cellT 是叶节点或者 cellT 的尺寸等于八叉树索引数组的栅格尺寸S'：

a) 获得 cellT 的最小坐标$(x_{min}, y_{min}, z_{min})$和最大坐标$(x_{max}, y_{max}, z_{max})$

b) nx1=$\langle(x_{min} - x_0)/S'\rangle$;　规定〈 〉表示四舍五入取整数

nx2=$\langle(x_{max} - x_0)/S'\rangle$;

ny1=$\langle(y_{min} - y_0)/S'\rangle$;　ny2=$\langle(y_{max} - y_0)/S'\rangle$;

nz1=$\langle(z_{min} - z_0)/S'\rangle$;　nz2=$\langle(z_{max} - z_0)/S'\rangle$;

c) 对八叉树索引数组中 i 从 nx1 到 nx2，j 从 ny1 到 ny2，k 从 nz1 到 nz2 的每一个栅格单元G_S[i][j][k]:

i. 令G_S[i][j][k]存放 cellT 的指针；

（2） 如果（1）的条件不满足，对于 cellT 的每一个子节点$subCell_{i,i=1,\ldots,8}$执行：

a) 迭代调用 GenerateSubGridOfRootCell($subCell_i$, G_S);

假设空间内有一点$P(x, y, z)$，利用 3.3.3 节介绍的算法生成了三维数组 G，并为 G 的每个栅格生成了八叉树，接下来给出应用了八叉树索引数组改进之后的定位叶节点的方法。

UpdatedLocateCell(P, G):

（1） 调用 3.4.2.1 节中的 LocateOctreeRootCell(P, G)，获得 P 所在的八叉树的根节点 rootCell;

（2） 获得根节点 rootCell 的最小坐标(x_0, y_0, z_0), 以及根节点对应的八叉树索引数组G_S，得到八叉树索引数组的栅格大小S'；

（3） nx=$\lfloor(x - x_0)/S'\rfloor$;　ny=$\lfloor(y - y_0)/S'\rfloor$;　nz=$\lfloor(z - z_0)/S'\rfloor$;

（4） 访问八叉树索引数组G_S，获得G_S[nx][ny][nz]处的节点指针 pCell，根据指针取得对应的节点 *pCell；

（5） 若*pCell 是叶节点，返回*pCell；若*pCell 不是叶节点，调用 3.4.2.1 节中的 LocateCellFromRoot(P, *pCell)，获得 P 所在的叶节点；

在编程实现过程中，我们规定了两个值：D_tree 和 M。假设每个八叉树根节点的尺寸是 gridCellSize，那么八叉树叶子节点的最小尺寸S_{min}=gridCellSize/2^{D_tree};而八叉树索引数组的栅格尺寸$S' = S_{min} \times 2^{M}$。

当 M=0 时，S'等于八叉树中最小的叶节点的尺寸，八叉树中每一个叶节点的指针都保存在了八叉树索引数组中，此时定位空间中一点 P 所在的叶节点时完全不需要从八叉树的根节点开始逐层比较坐标，因此定位叶节点的速度较快，但是此时八叉树索引数组较大，占用内存增加。

当 M= D_tree 时， S'等于八叉树的根节点尺寸 gridCellSize，定位点 P 所在的叶节点要从八叉树的根节点开始逐层比较坐标，因此速度较慢。我们可以通过调节 D_tree 和 M 的值，在内存占用和执行速度之间找到平衡。

第四章 随机行走算法的实现

§4.1 高斯面的生成与点的选取[10]

根据图 2.2 所示的算法，在选定了以某个导体作为出发点进行随机行走之后，需要构造包围该导体的高斯面。构造高斯面最直观的思路是产生一个实际的多面体的将导体包围住，这个多面体每个侧面都是一个多边形，每个多边形由所有拐点处的坐标来定义。当需要从高斯面上按照均匀分布选出一个点时，首先根据高斯面上各个侧面的面积决定的概率选定某个侧面，然后在该侧面上按照均匀分布选出一个点。

上述方法在理论上是可行的，然而存在两个问题：

（1） 生成多面体的算法较复杂。在版图中一个完整导体的形状较为复杂，如图 3.5(b)所示的导体，很难生成一个实际的多面体将该导体包裹起来；

（2） 从高斯面上选点的算法较为复杂。假设生成了一个高斯面将图 3.5(b)所示的导体包围起来，那么这个高斯面会包含很多个侧面，同时每个侧面往往不是规则的长方形，而是不规则的多边形。这使得从高斯面的某个侧面上随机选取一个点的算法较为复杂。

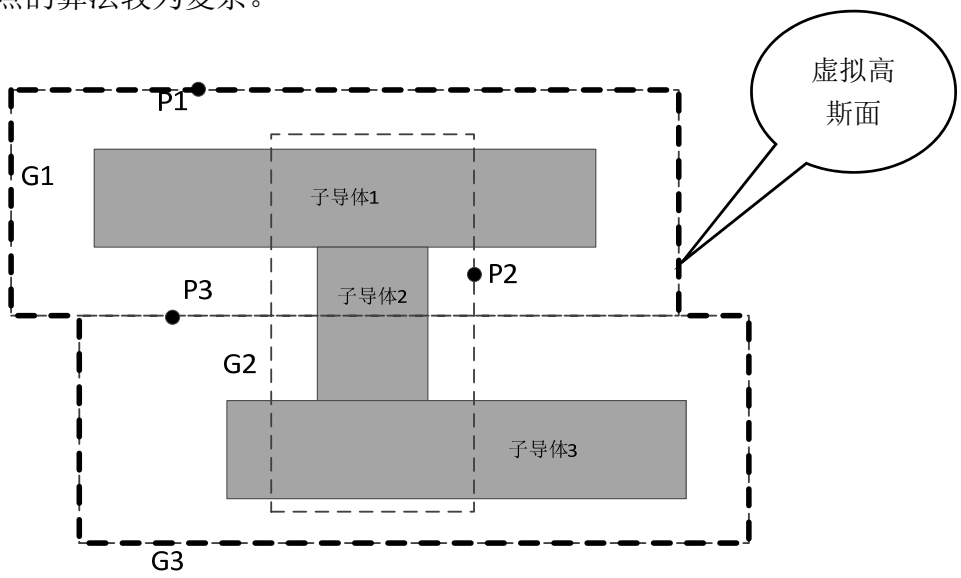


图 4.1 生成虚拟高斯面的平面示意图

基于上述两点，[10]提出了一种“虚拟高斯面”的方法，该方法适用于构建复杂形状导体的高斯面。如图 4.1，假设存在一种工字形导体，通过 3.1 节介绍的导体分割的方法可以将该导体分割成三个子导体，每个子导体都是一个长方体；为每一个子导体构造一个高斯面（为了表述简便本文也称子导体的高斯面为子高斯面）。如图 4.1 中为子导体 1 生成子高斯面G_1，为子导体 2 生成子高斯面G_2，为子导体 3 生成

子高斯面G_3，需要注意的是每个子高斯面可能会穿过其他附近的子导体。这样就完成了为该工字形导体构造虚拟高斯面的过程。

当需要从该工字形导体的虚拟高斯面上选取出一点时，首先从该导体的所有子导体对应的子高斯面中选出一个子高斯面，然后从选出的子高斯面上再选出一个点P，然后需要判断点 P 的位置：若点 P 处在某个子导体的高斯面内部，如图 4.1 中的P_2和P_3，那么说明点 P 并不在工字形导体的虚拟高斯面上，因此丢弃点 P，重新进行选择点；若点 P 不处于任何子导体高斯面的内部，如图 4.1 中的P_1，那么说明点 P 处在虚拟高斯面上，此时成功地从虚拟高斯面的表面选出了一点。

接下来进一步讨论这一过程的细节。

§4.1.1 高斯面的生成

生成高斯面的原则是使随机行走的跳转次数尽可能减少，根据第二章和第三章对随机行走以及生成最大转移立方体的分析，要使得跳转次数尽量少，需要随机行走能够尽快跳转到某个导体的表面。据此可以确定生成子导体高斯面的一个方法：首先规定一组方向$\{PX, PY, PZ, NX, NY, NZ\}$，分别表示空间内 x 轴正方向、y 轴正方向、z 轴正方向、x 轴负方向、y 轴负方向、z 轴负方向；在生成子导体$subCond_i$的高斯面时，分别计算该子导体在其 6 个面所对应的方向上到最近（但不直接接触）的子导体块的距离$D[i], i=PX, PY, PZ, NX, NY, NZ$，将$D[i]/2$作为子导体在各个面上向外扩展的距离，即生成子导体高斯面的位置。

如图 4.2 所示，若要为子导体 1 生子成高斯面，需要计算在 6 个方向上与子导体 1 最近（但不接触的）子导体的距离。例如，在子导体 1 的右侧距离子导体 1 最近的是子导体 k，二者之间的距离是D_R，取$D_R/2$处（点 M 处）放置子导体 1 的高斯面G_1。其他方向上同理，需要注意的是子导体 1 下方与子导体 1 距离最近并且不接触的是子导体 3，而不是子导体 2。

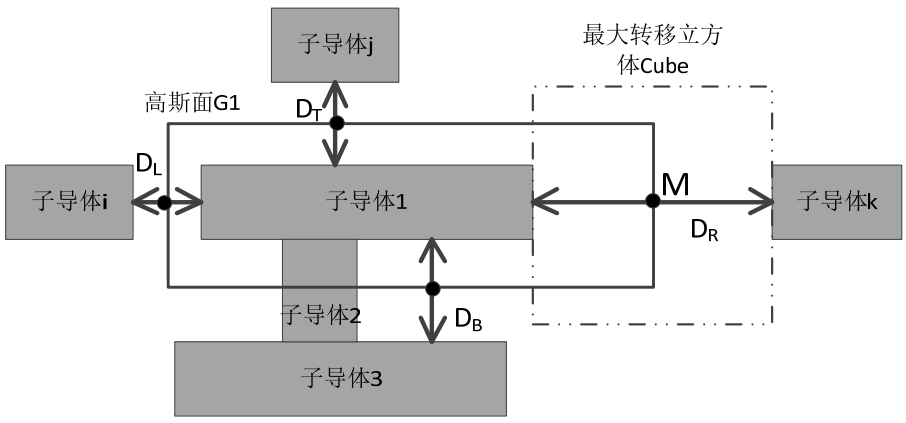


图 4.2 生成高斯面算法示平面意图

将$D[i]/2$作为放置高斯面的位置的优缺点：

优点：

以高斯面上的点为中心做出的最大转移立方体至少与两个子导体的表面接触；可以增加随机行走第一次跳转就落在导体表面的概率。

缺点：

子导体在产生高斯面时每个面向外扩展的距离不同，导体的高斯面在不同区域距离导体表面的距离差异很大。那么以高斯面上的点为中心做出的第一个最大转移立方体的棱长 L 的方差较大。根据 2.2.2 节的分析并结合格林函数的表达式[6]可知，权重函数$w(\vec{r},\vec{r}_1)$的表达式(2.2.11)中含有因式$1/L$，因此L方差较大会导致权重值的方差也较大，从而会增加随机行走的收敛时间。

为解决上述问题，[10]提出了采用一个缩放因子$scale$来调节，基本算法如下：

（1） 计算得到子导体 6 个面上高斯面向外扩展的距离

$\{D[PX],D[PY],D[PZ],D[NX],D[NY],D[NZ]\}$；

（2） 在$D[i]$,$(i=PX,PY,PZ,NX,NY,NZ)$中找到最小值，记为$minDis$；

（3） 对于每一个$D[i]$，比较$D[i]$与$minDis\times scale$的大小，较小者作为新的$D[i]$的值；

（4） 将更新后的$D[i]$作为生成高斯面时向外膨胀的距离。

通过该算法可知，当缩放因子为 1 时，高斯面向外膨胀的尺寸选定为初始$\{D[PX],D[PY],D[PZ],D[NX],D[NY],D[NZ]\}$中的最小值[8,10]；当缩放因子为正无穷时，那么相当于没有对原始膨胀尺寸进行修改。

高斯面生成的一般原则是**导体的各处的高斯面到导体表面的距离不能波动过大；导体的高斯面不能离导体表面过近。**原因如下：

（1） 当缩放因子过大时，对高斯面到导体表面的距离起不到调控的效果，高斯面上的不同点到导体表面的距离波动较大，根据上面的分析可知此时权重函数值的方差较大，因而会增加随机行走的收敛时间；

（2） 当缩放因子过小时，高斯面上的点到导体表面的平均距离过近，随机行走第一次跳转就终结在自身导体表面的概率增加，同样会导致随机行走的收敛速度变慢

此外，在测试中发现，随着缩放因子的减小，平均每次随机行走的时间逐渐减少。产生这个现象的原因是，缩放因子减小使得高斯面到导体表面的平均距离减小，第一个最大转移立方体与高斯面所包围的导体的接触面占比增加，随机行走第一次跳转终结在自身导体表面的概率增加，平均每次随机行走的跳转次数减少，因此平均每次随机行走的时间也会减少。

§ 4.1.2 在高斯面上选取点

在高斯面上按照均匀分布选取点时，首先根据父导体的子高斯面列表中的每个子高斯面的面积所决定的概率

$p_i = \frac{A_{subG_i}}{\sum A_{subG_k}}$，$A_{subG_i}$表示高斯面 i 的面积，$\sum A_{subG_k}$表示高斯面总面积

抽取出一个子高斯面；然后同样根据面积决定的概率从子高斯面的六个面中选出一个面。然后在选中的面上按照均匀分布选择一个点 P。点 P 在高斯面上的分布情况如图 4.4 所示，可以分成四种情况：

（1） 点 P 在导体的虚拟高斯面表面，如点 p1，那么从虚拟高斯面上选点成功，保留点 P;

（2） 点 P 处在某个子高斯面内部，如点 p2，那么将点 P 舍弃，重新选择点；

（3） 点 P 同时处在两个（或两个以上）的子高斯面表面，并且存在两个子高斯面在该点处的外法向量方向相反，如点 p3，那么将点 P 舍弃，重新选择点；

（4） 点 P 同时处在 N 个子高斯面的表面，并且这 N 个子高斯面在该点处的外法向量方向相同，如点 p4，那么点 P 有 1/N 的概率的保留。

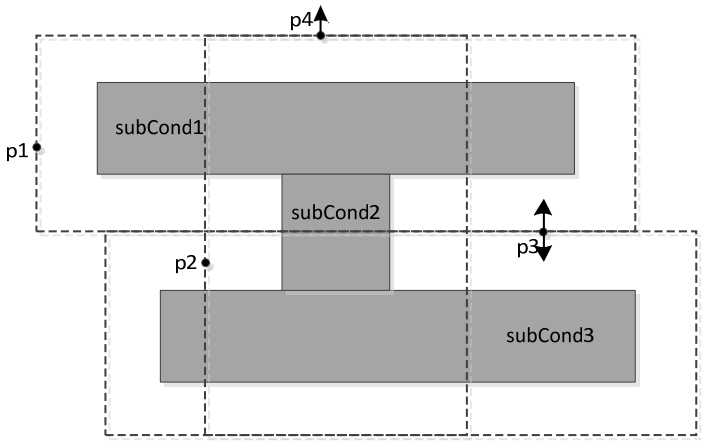


图 4.3 在虚拟高斯面上选择点的示意图

在分析从子高斯面 i 上选出的点 P 符合图 4.4 中 p1-p4 的哪种情况时，只需要检查所有与子高斯面 i 相邻的其它子高斯面即可。因此可以预先计算每个子高斯面相邻的子高斯面，并为每个子高斯面生成一个相邻子高斯面列表(neighborList)。在判断从某个子高斯面上选出的点的位置时，只需要考虑该子高斯面的 neighborList 中的元素即可。

根据上述分析，可以给出在某个导体$cond_i$的高斯面表面按照均匀分布选取点的算法流程图。

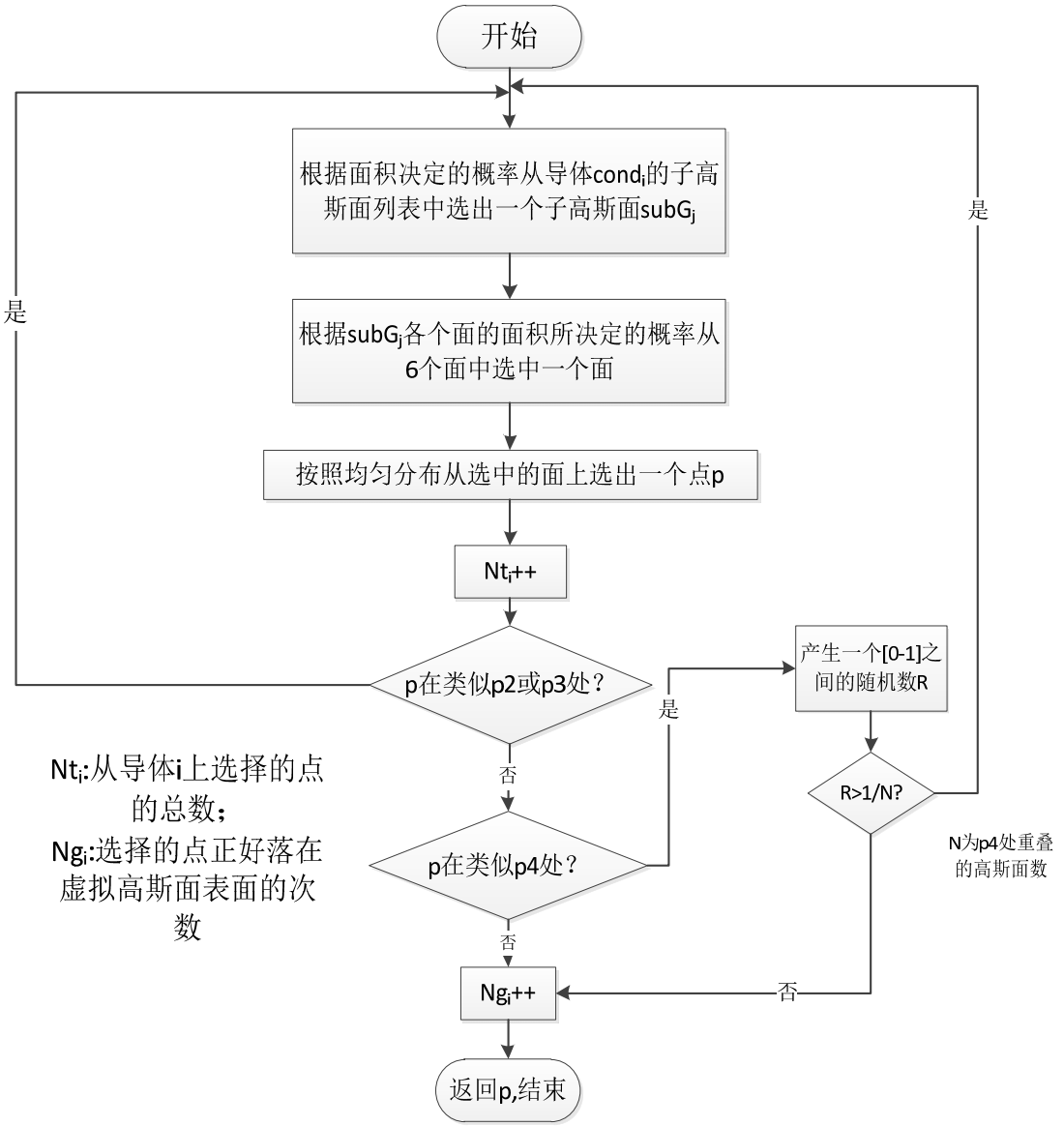


图 4.4 从高斯面上选点算法流程图

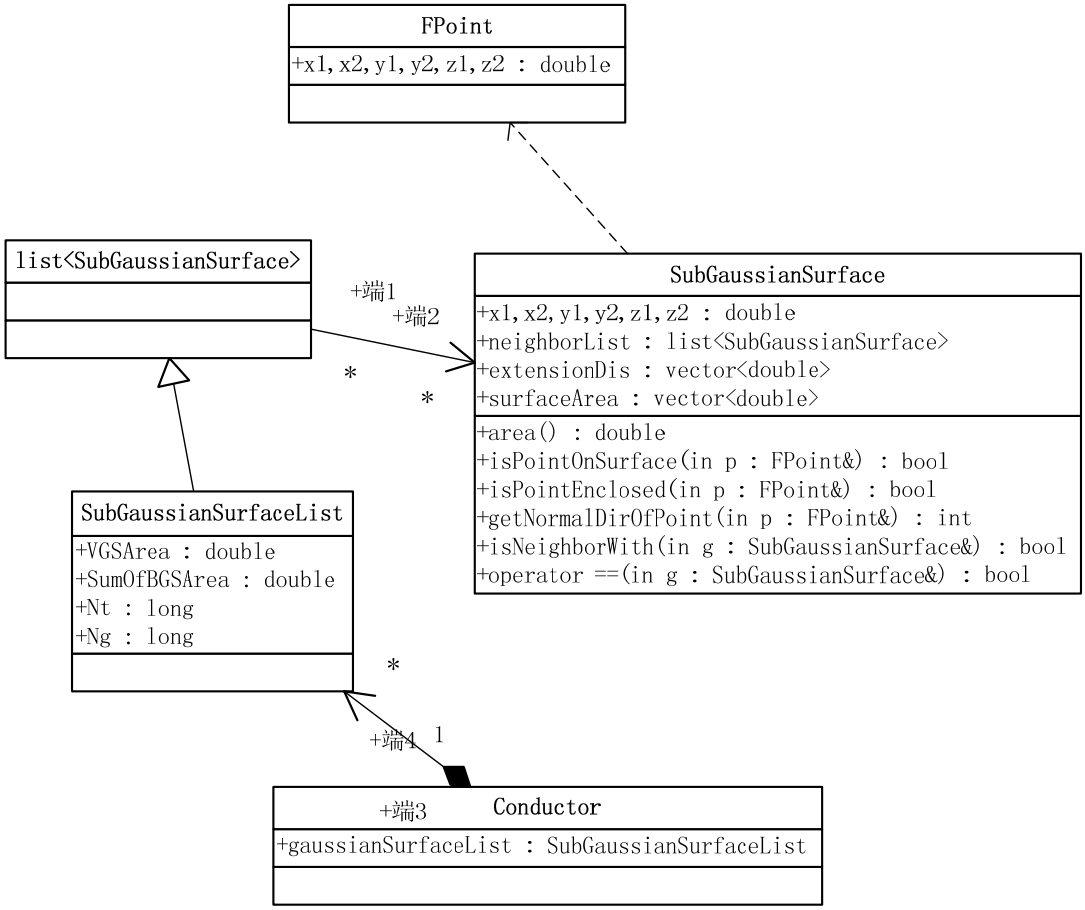
根据图 4.4 所示的算法，导体$cond_i$的虚拟高斯面的面积

$$A_{G_i} = \frac{N_{g_i}}{N_{t_i}} \times \sum A_{subG_k} \tag{4.1.1}$$

其中$\sum A_{subG_k}$表示导体$cond_i$的子高斯面的面积和。

§4.1.3 编程实现

在实现上述算法过程中涉及到的关键的数据结构是 SubGaussianSurface 类，这里给出相关数据结构的描述图。



FPoint 浮点类	
double x1, y1, z1	空间内一点的三维坐标
SubGaussianSurface 子高斯面类	
double x1,x2,y1,y2,z1,z2	子高斯面体对角线的顶点坐标
list<SubGaussianSurface> neighborList	相邻子高斯面列表
vector<double> extensionDis	子高斯面在六个方向上相对于子导体表面向外膨胀的距离
vector<double> surfaceArea	子高斯面的表面积
double are ()	返回子高斯面的表面积
bool isPointOnSurface (Fpoint &p)	判断点 p 是否在子高斯面的表面
bool isPointEnclosed (Fpoint &p)	判断点 p 是否被该子高斯面包围
int getNormalDirOfPoint (Fpoint &p)	(1) 若 p 不在子高斯面的表面，返回 0； (2) 若 p 在子高斯面的表面，判断子高斯面在 p 处的外法向量的方向 D,

	D∈{PX,PY,PZ,NX,NY,NZ}; 根据 D 的值确定返回值 D==PX, return 1; D==NX, return -1; D==PY, return 2; D==NY, return -2; D==PZ, return 3; D==NZ, return -3. ∃子高斯面SG_i和SG_j，满足 SG_i. getNormalDirOfPoint (p)!=0 && SG_i. getNormalDirOfPoint (p)+ SG_j. getNormalDirOfPoint (p)==0， 那么点 p 处于类似于图 3.8 中 p3 的位置。
bool isNeighborWith (SubGaussianSurface &SG)	判断子高斯面 SG 是否与当前子高斯面相邻，该函数用于为每个子高斯面生成 neighborList 时判断子高斯面是否相邻。
operator ==(SubGaussianSurface &SG)	判断子高斯面 SG 是否与当前子高斯面是同一个子高斯面，该函数用于为每个子高斯面生成 neighborList 时避免把调用者自身也添加进列表。
SubGaussianSurfaceList : public list<SubGaussianSurface> 子高斯面列表类	
double SumOfBGSArea	子高斯面列表中所有子高斯面的表面积之和
long Nt	随机行走过程中从该子高斯面列表中选取的点的总数
long Ng	随机行走过程中从该子高斯面列表中成功选取的点的总数。这里成功选取指的是选取的点刚好在虚拟高斯面表面并且作为一次随机行走的起始点。
double VGSArea	导体的虚拟高斯面的表面积。 VGSArea= SumOfBGSArea × Ng / Nt
Conductor 导体类	
SubGaussianSurfaceList gaussianSurfaceList	导体的子高斯面列表，每一个导体都维护一个子高斯面列表，用于随机行走时从该导体的虚拟高斯面上选择起始点。

图 4.5 与导体分割与整合相关的数据结构及成员描述

§4.2 表面格林函数和权重函数的计算

§4.2.1 相关公式[6]

假设一个立方体的体对角线上顶点的坐标为(0,0,0)和(L,L,L)，那么对于该立方体内的某一点$\vec{r}(x,y,z)$处的电势$\phi(\vec{r})$，在立方体顶部$\vec{r}'(x',y',L)$处的表面格林函数的表达式为[6]

$$G_{\phi PZ}(x',y',L|x,y,z)=\frac{4}{L^2}\sum_{n_x=1}^{\infty}\sum_{n_y=1}^{\infty}\sin\left(\frac{\pi n_x}{L}x\right)\sin\left(\frac{\pi n_y}{L}y\right)\sinh\left(\frac{\pi n_z}{L}z\right)\times\frac{sin\{(\pi n_x/L)x'\}\sin\{(\pi n_y/L)y'\}}{\sinh(\pi n_z)} \tag{4.2.1}$$

其中$n_z=\sqrt{{n_x}^2+{n_y}^2}$，$PZ$表示法向量指向 Z 轴正方向，可参考 3.2.1 节的定义。在其他 5 个面上的格林函数值均可以用(3.3.1)式表示：

$$\begin{cases}G_{\phi PX}(L,y',z'|x,y,z)=G_{\phi PZ}(y',z',L|y,z,x)\\ G_{\phi PY}(x',L,z'|x,y,z)=G_{\phi PZ}(x',z',L|x,z,y)\\ G_{\phi NX}(0,y',z'|x,y,z)=G_{\phi PZ}(y',z',L|y,z,L-x)\\ G_{\phi NY}(x',0,z'|x,y,z)=G_{\phi PZ}(x',z',L|x,z,L-y)\\ G_{\phi NZ}(x',y',0|x,y,z)=G_{\phi PZ}(x',y',L|x,y,L-z)\end{cases} \tag{4.2.2}$$

$G_{\phi PX}(L,y',z'|x,y,z)=G_{\phi PZ}(y',z',L|y,z,x)$的含义：$G_{\phi PX}(L,y',z'|x,y,z)$的表达式是用$y',z'$分别替代(4.2.1)式中的$x',y'$；$y,z,x$分别替代(4.2.1)式中的$x,y,z$。

其余各式同理。

根据式(2.2.11)可知权重函数中需要求格林函数的负梯度

$$-\nabla_r G_{\phi PZ}(x',y',L|x,y,z)=-\frac{4}{L^3}\sum_{n_x=1}^{\infty}\sum_{n_y=1}^{\infty}\begin{bmatrix}\pi n_x\cos\left(\frac{\pi n_x}{L}x\right)\sin\left(\frac{\pi n_y}{L}y\right)\sinh\left(\frac{\pi n_z}{L}z\right)\\ \pi n_y\sin\left(\frac{\pi n_x}{L}x\right)\cos\left(\frac{\pi n_y}{L}y\right)\sinh\left(\frac{\pi n_z}{L}z\right)\\ \pi n_z\sin\left(\frac{\pi n_x}{L}x\right)\sin\left(\frac{\pi n_y}{L}y\right)\cosh\left(\frac{\pi n_z}{L}z\right)\end{bmatrix}\times\frac{sin\{(\pi n_x/L)x'\}\sin\{(\pi n_y/L)y'\}}{\sinh(\pi n_z)} \tag{4.2.3}$$

利用(4.2.2)式可以求出在其它 5 个面上格林函数的负梯度值，可以预见它们可以用(4.2.3)式表示。因为在随机行走算法中是以点$\vec{r}$为中心产生的最大转移立方体，所以这里将$x=y=z=L/2$代入(4.2.1)式，可得

$$L^2G_{\phi PZ}(x',y',L)=2\sum_{n_x=1}^{\infty}\sum_{n_y=1}^{\infty}\frac{\sin\left(\frac{\pi n_x}{2}\right)\sin\left(\frac{\pi n_y}{2}\right)}{\cosh(\pi n_z/2)}sin\left(\pi n_x\frac{x'}{L}\right)\sin\left(\pi n_y\frac{y'}{L}\right) \tag{4.2.4}$$

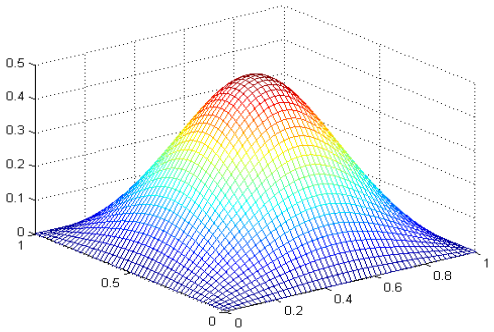
为表述简便首先定义如下缩略形式：

$$L^2G_{\phi PZ}(a,b) = L^2G_{\phi PZ}(x',y',L)|x'=a,y'=b$$

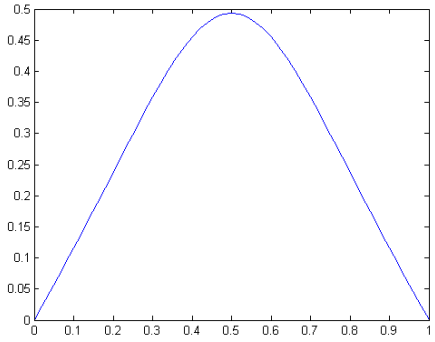
将$x=y=z=L/2$代入(4.2.2)式，可得

$$\begin{cases} L^2G_{\phi PX}(L,y_0',z_0') = L^2G_{\phi PZ}(y_0',z_0') \\ L^2G_{\phi PY}(x_0',L,z_0') = L^2G_{\phi PZ}(x_0',z_0') \\ L^2G_{\phi NX}(0,y_0',z_0') = L^2G_{\phi PZ}(y_0',z_0') \\ L^2G_{\phi NY}(x_0',0,z_0') = L^2G_{\phi PZ}(x_0',z_0') \\ L^2G_{\phi NZ}(x_0',y_0',0) = L^2G_{\phi PZ}(x_0',y_0') \end{cases} \tag{4.2.5}$$

所以在计算其它 5 个面上的格林函数值时，只需要按照(4.2.5)式所示的坐标映射关系将表面上的坐标值代入$L^2G_{\phi PZ}(x',y',L)$即可。



a. $G_{\phi PZ}$图像 L=1, x,y ∈ [0,1]



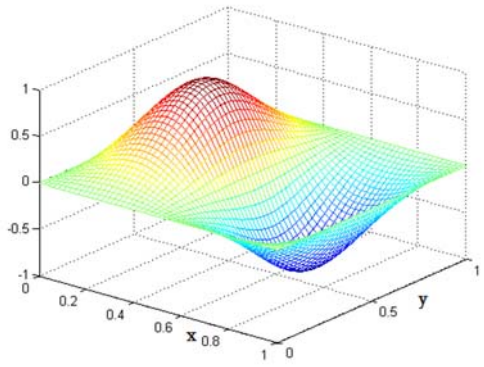
b. $G_{\phi PZ}$图像 L=1, x=1/2 ,y ∈ [0,1]

图 4.6 PZ 方向的格林函数图像

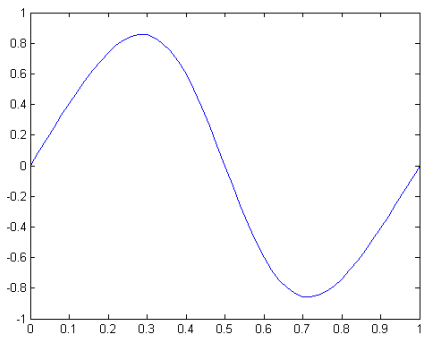
将$x=y=z=L/2$代入(4.2.3)式，可得

$-L^3\nabla_r G_{\phi PZ}(x',y',L) =$

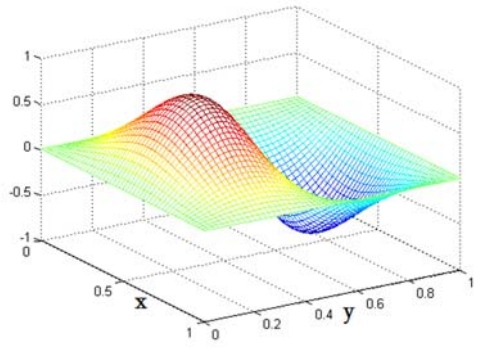
$$-2\pi\sum_{n_x=1}^{\infty}\sum_{n_y=1}^{\infty}\begin{bmatrix} \dfrac{n_x\cos(\pi n_x/2)\sin(\pi n_y/2)}{\cosh(\pi n_z/2)} \\ \dfrac{n_y\sin(\pi n_x/2)\cos(\pi n_y/2)}{\cosh(\pi n_z/2)} \\ \dfrac{n_z\sin(\pi n_x/2)\sin(\pi n_y/2)}{\sinh(\pi n_z/2)} \end{bmatrix} \times sin\left(\pi n_x\frac{x'}{L}\right)\sin\left(\pi n_y\frac{y'}{L}\right) \tag{4.2.6}$$



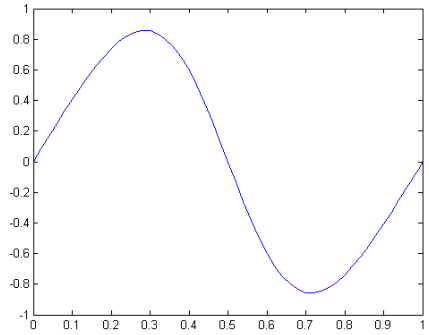
a. $-\partial G_{\phi PZ}/\partial$x, L=1, x,y ∈ [0,1]



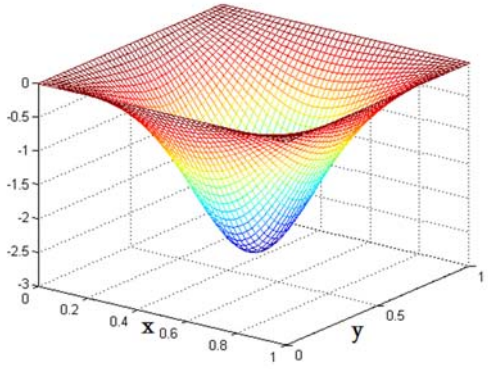
b. $-\partial G_{\phi PZ}/\partial$x, y=1/2 ,x ∈ [0,1]



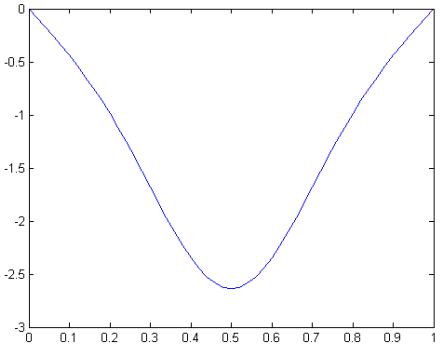
c. $-\partial G_{\phi PZ}/\partial y,\ L=1,\ x,y\in[0,1]$



d. $-\partial G_{\phi PZ}/\partial y,\ x=1/2\ ,y\in[0,1]$



e. $-\partial G_{\phi PZ}/\partial z,\ L=1,\ x,y\in[0,1]$



f. $-\partial G_{\phi PZ}/\partial z,\ x=1/2\ ,y\in[0,1]$

图 4.7 PZ 方向的格林函数负梯度图像

为了表述简便首先定义如下缩略表达形式：

$$-L^3\frac{\partial G_{\phi PZ}}{\partial z}(a,b)=-L^3\frac{\partial G_{\phi PZ}}{\partial z}(x',y',L)|x'=a,y'=b$$

令$x=y=z=L/2$，可以得到在其它面上表面格林函数的负梯度值

$$\begin{cases}
-L^3\frac{\partial G_{\phi PX}}{\partial x}\left(L,y_0',z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial z}\left(y_0',z_0'\right);\ -L^3\frac{\partial G_{\phi PX}}{\partial y}\left(L,y_0',z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial x}\left(y_0',z_0'\right);\ -L^3\frac{\partial G_{\phi PX}}{\partial z}\left(L,y_0',z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial y}\left(y_0',z_0'\right);\\
-L^3\frac{\partial G_{\phi PY}}{\partial x}\left(x_0',L,z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial x}\left(x_0',z_0'\right);\ -L^3\frac{\partial G_{\phi PY}}{\partial y}\left(x_0',L,z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial z}\left(x_0',z_0'\right);\ -L^3\frac{\partial G_{\phi PY}}{\partial z}\left(x_0',L,z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial y}\left(x_0',z_0'\right);\\
-L^3\frac{\partial G_{\phi NX}}{\partial x}\left(0,y_0',z_0'\right)=L^3\frac{\partial G_{\phi PZ}}{\partial z}\left(y_0',z_0'\right);\ -L^3\frac{\partial G_{\phi NX}}{\partial y}\left(0,y_0',z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial x}\left(y_0',z_0'\right);\ -L^3\frac{\partial G_{\phi NX}}{\partial z}\left(0,y_0',z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial y}\left(y_0',z_0'\right);\\
-L^3\frac{\partial G_{\phi NY}}{\partial x}\left(x_0',0,z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial x}\left(x_0',z_0'\right);\ -L^3\frac{\partial G_{\phi NY}}{\partial y}\left(x_0',0,z_0'\right)=L^3\frac{\partial G_{\phi PZ}}{\partial z}\left(x_0',z_0'\right);\ -L^3\frac{\partial G_{\phi NY}}{\partial z}\left(x_0',0,z_0'\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial y}\left(x_0',z_0'\right);\\
-L^3\frac{\partial G_{\phi NZ}}{\partial x}\left(x_0',y_0',0\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial x}\left(x_0',y_0'\right);\ -L^3\frac{\partial G_{\phi NZ}}{\partial y}\left(x_0',y_0',0\right)=-L^3\frac{\partial G_{\phi PZ}}{\partial y}\left(x_0',y_0'\right);\ -L^3\frac{\partial G_{\phi NZ}}{\partial z}\left(x_0',y_0',0\right)=L^3\frac{\partial G_{\phi PZ}}{\partial z}\left(x_0',y_0'\right);
\end{cases}$$

(4.2.7)

根据(2.2.11)和(4.2.4)-(4.2.7)式可以计算权重函数的值。

§4.2.2 离线数据表的实现

观察(4.2.4)式和(4.2.6)式，等号右边的值取决于x'和y'与L的比值而与x'和y'的绝对大小无关，并且$x'/L, y'/L \in [0,1]$。因此可以预先计算格林函数以及格林函数的负梯度存储在离线文件中，在随机行走之前将离线文件读取进内存，这样可以避免在随机行走过程中实时计算，加快随机行走的速度。思路如下：

（1） 设L=1，把坐标系中(0,0)到(1,1)的正方形区域平均分割成$N \times N$的栅格。

（2） 在每个小栅格内随机选取M个点(x_i, y_i), i=1,2, … M ，将x_i, y_i代入(4.2.4)式和(4.2.6)式计算格林函数值和格林函数的负梯度值。取M个计算结果的平均值作为单位立方体条件下该小栅格内格林函数和格林函数的负梯度的估计值。

（3） 将计算的数据以二维矩阵的形式存放在文件中，在随机行走执行之前从文件中将二维矩阵读取进二维数组中。

（4） 在随机行走过程中，已知最大转移立方体的棱长为L_t，如果要计算在表面上某一点(x', y', L_t)的格林函数值和格林函数的梯度值，需要计算该点在二维数组中对应的位置$[n_x, n_y]$

$$\begin{cases} n_x = \min\left(\left\lfloor \dfrac{x'/L_t}{1/N} \right\rfloor , N-1\right) \\ n_y = \min\left(\left\lfloor \dfrac{y'/L_t}{1/N} \right\rfloor , N-1\right) \end{cases} \tag{4.2.8}$$

在二维数组的$[n_x, n_y]$处分别取得单位立方体条件下格林函数值和格林函数负梯度值，然后再进行进一步处理。

使用离线数据表的好处是避免了对格林函数和权重函数的实时计算，极大的提高的速度，但是由于在随机行走时需要将离线数据表读进二维数组，因此对内存的占用增加。本文提取一种可行的改进方法，通过观察图 4.7 和图 4.8，我们发现还可以利用表面格林函数及其负梯度的对称性进一步减小离线数据表的大小。以表面格林函数的表达式(4.2.4)式为例，

因为 $sin\left(\pi n_x \frac{\frac{L}{2}-x'}{L}\right) = \sin\left(\frac{\pi n_x}{2} - \frac{\pi n_x x'}{L}\right) = \sin\left(\frac{\pi n_x}{2}\right)\cos\left(\frac{\pi n_x x'}{L}\right)$,

$sin\left(\pi n_x \frac{\frac{L}{2}+x'}{L}\right) = \sin\left(\frac{\pi n_x}{2} + \frac{\pi n_x x'}{L}\right) = \sin\left(\frac{\pi n_x}{2}\right)\cos\left(\frac{\pi n_x x'}{L}\right)$, $\cos\left(\frac{\pi n_x}{2}\right) = 0$;

所以 $sin\left(\pi n_x \frac{\frac{L}{2}-x'}{L}\right) = sin\left(\pi n_x \frac{\frac{L}{2}+x'}{L}\right)$;

同理 $sin\left(\pi n_y \frac{\frac{L}{2}-y'}{L}\right) = sin\left(\pi n_y \frac{\frac{L}{2}+y'}{L}\right)$;

因此$L^2G_{\phi PZ}\left(\frac{L}{2}-x',y',L\right)=L^2G_{\phi PZ}\left(\frac{L}{2}+x',y',L\right)$

$$L^2G_{\phi PZ}\left(x',\frac{L}{2}-y',L\right)=L^2G_{\phi PZ}\left(x',\frac{L}{2}+y',L\right)$$

即$L^2G_{\phi PZ}(x',y',L)$关于$x'=L/2$和$y'=L/2$对称。

同理，可以分析得到：

（1） $-\partial G_{\phi PZ}/\partial$x关于$x'=L/2$负对称，关于$y'=L/2$对称；

（2） $-\partial G_{\phi PZ}/\partial$y关于$x'=L/2$对称，关于$y'=L/2$负对称；

（3） $-\partial G_{\phi PZ}/\partial$z关于$x'=L/2$对称，关于$y'=L/2$对称；

利用上述对称关系可以将存放格林函数值以及格林函数负梯度值的二维数组仅保留$(0,0)$到$(L/2,L/2)$的矩形范围内的数据，即原数组的 $1/4$, 因此可以节省内存占用。

§4.2.3 拒绝性采样

随机行走的重要步骤是在最大转移立方体的表面上按照表面格林函数所决定的概率分布选出一个点，作为随机行走下一步跳转到的位置。产生符合表面格林函数决定的分布的方法是拒绝性采样。拒绝性采样的基本原理如下：

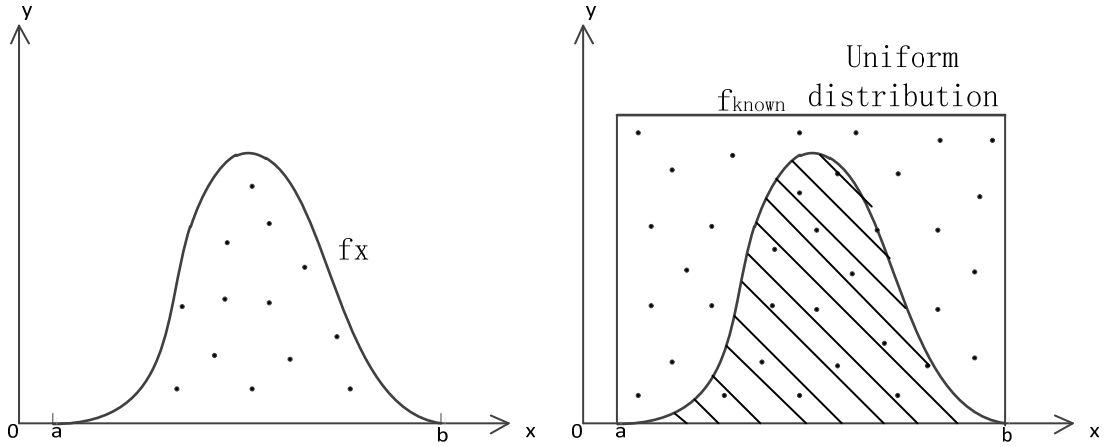
如图 4.9 a 所示，假设分布函数为$f_x(x),x\in[\mathrm{a,b}]$，如果要生成一个符合$f_x$的分布，可以选择一个分布函数$f_{known}(x),x\in[\mathrm{a,b}]$，并且满足

$$f_{known}(x)\geq f_x(x),x\in[\mathrm{a,b}]$$

首先根据$f_{known}(x)$决定的分布采样得到点$p_i(x,y)$，通过比较p_i的坐标(x,y)与$f_x(x)$的关系来决定是否保留p_i。若$y>f_x(x)$，舍弃p_i；$y\leq f_x(x)$，则保留p_i（如图 4.8b 中的阴影区域）。一般$f_{known}(x)$对应一个已知的分布，如使用

$$f_{known}(x)=\text{constant},\ \text{constant}>=\max\left(f_x(x)\right),\ x\in[\mathrm{a,b}]$$

表示在区间[a,b]内的均匀分布。



a. 要生成的分布对应的分布函数　　b. 利用已知分布生成目标分布

图 4.8 拒绝性采样原理示意图

假设已知最大转移立方体体对角线上顶点坐标为(x_0, y_0, z_0)和(x_1, y_1, z_1)，棱长 $L = x_1 - x_0$。利用 4.2.2 节所介绍的方法利用二维数组存放表面格林函数值，下面给出算法流程图。

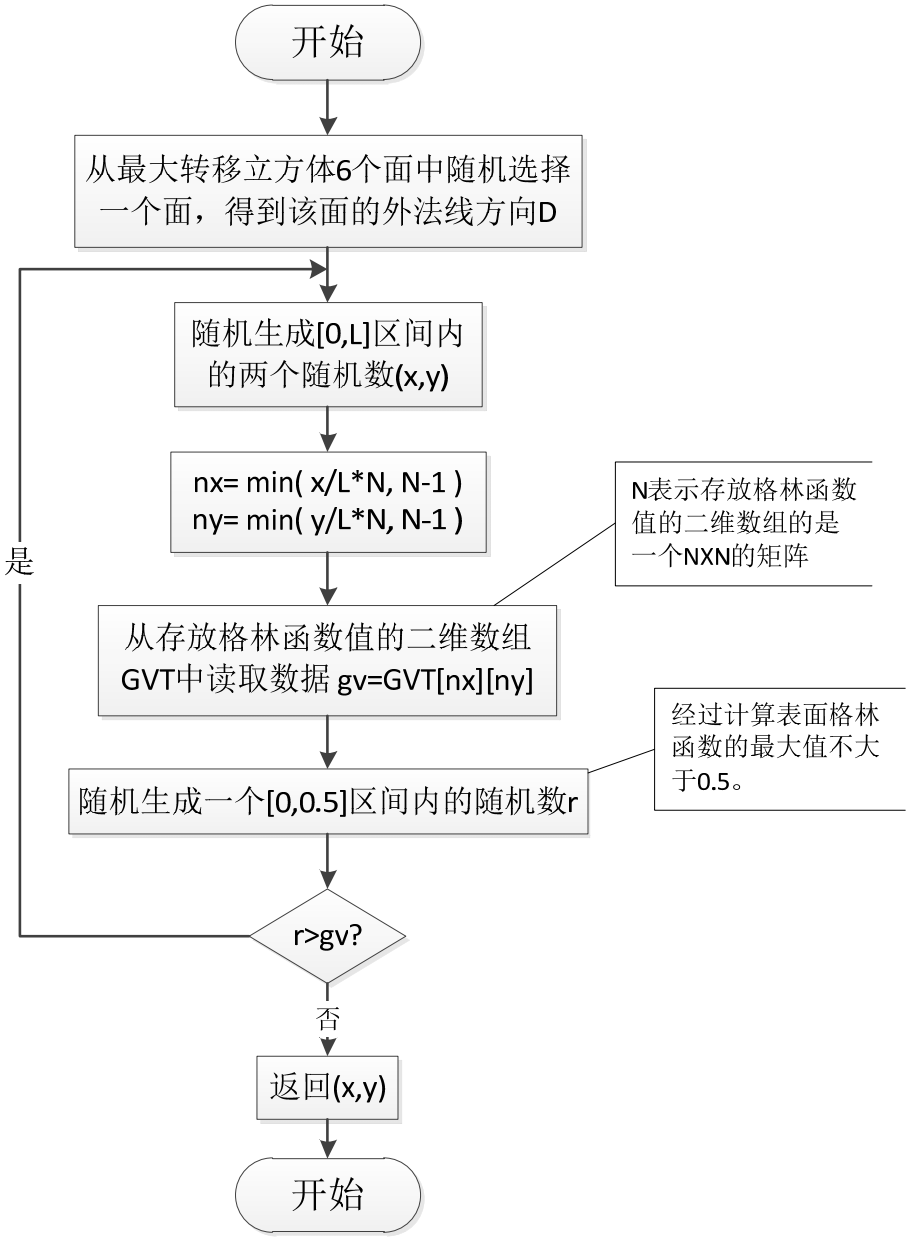


图 4.9 生成格林函数所决定的分布算法流程图

§4.3 随机行走算法的并行实现

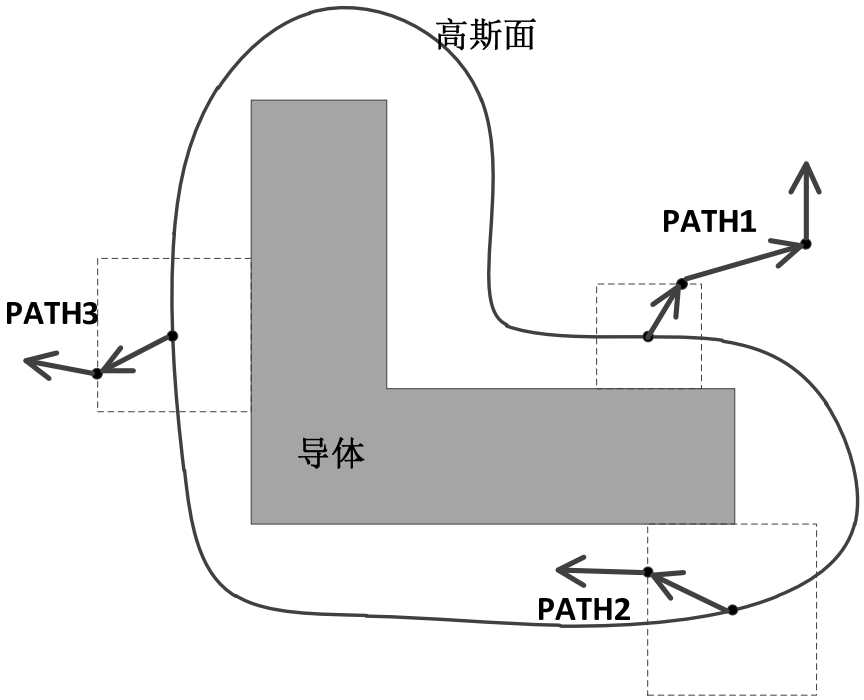


图 4.10 随机行走并行部分示意图

由于每次随机行走彼此之间是独立的，因此随机行走算法非常适合通过并行计算加速[8]。我们把从导体的高斯面上选一个点开始，进行若干次跳转，最后终结在导体表面或者跳出边界的过程称为一次随机行走（PATH）。如图 4.10 所示，不同的 PATH 之间，如 PATH1、PATH2、PATH3 之间是相互独立的，只有当 PATH 终结时，需要访问全局变量更新电容矩阵的值（**参考 2.2 节的随机行走算法的原理分析**）。因此，可以将 PATH 的过程进行并行执行。

随机行走一次 PATH 的算法流程图如图 4.11 所示。

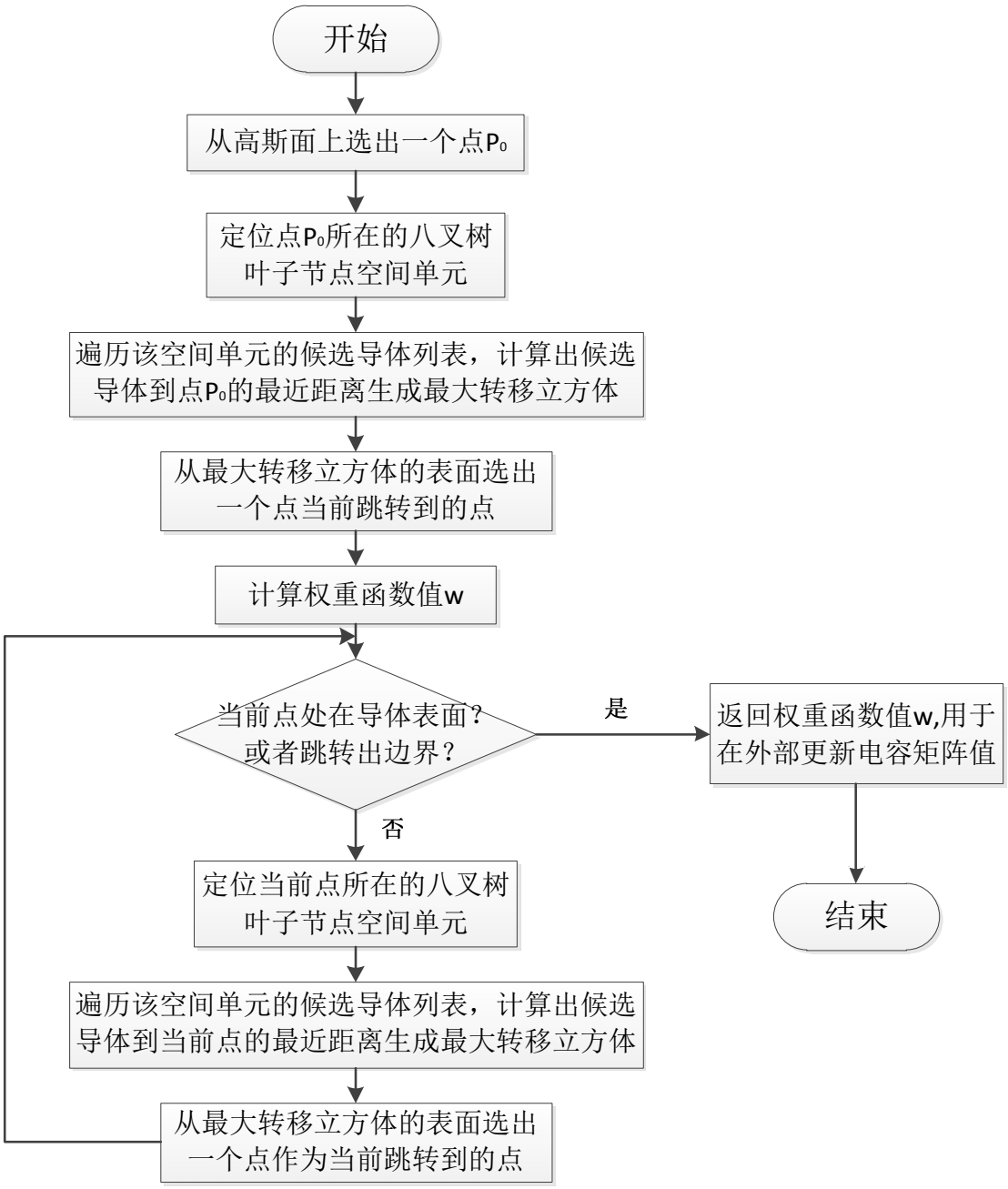


图 4.11 PATH 的算法流程图

在实现多线程算法时时，为了避免多个线程频繁的更新全局电容矩阵导致线程冲突，可以为每个线程设置一个本地的电容矩阵。在每个线程内部，每执行一次 PATH 算法都更新一次本地的电容矩阵。在执行 n（n=10000）次随机行走的 PATH 算法之后，将本地的电容矩阵中存储的值更新到全局的电容矩阵中。本文将这一算法称为随机行走核心算法。该算法的流程图如图 4.12 所示：

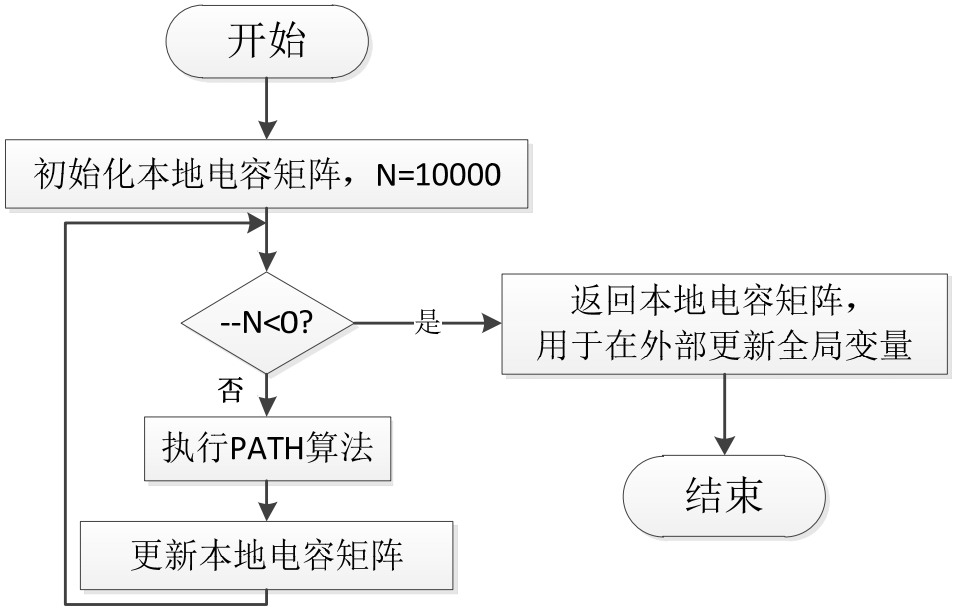


图 4.12 随机行走核心算法流程图

根据上述分析，可以给出采用完整的随机行走并行实现算法的流程图。如图 4.13 所示：

（1） 读取版图信息，对导体进行切割与整合，生成导体列表；

（2） 然后执行基于八叉树结构的空间管理算法，并采用 3.4 节讨论的加速技术；

（3） 为导体构造高斯面；

（4） 加载离线数据表并初始化全局变量；

（5） 创建多个线程执行随机行走的核心算法，即图 4.12 所示的随机行走算法。在每个线程中执行 n 次随机行走（n=10000）之后，更新全局变量的值。（要注意每个线程在更新全局变量时为保证线程安全需要对全局变量加锁，当更新完成后再释放锁。）

（6） 检测随机行走的次数是否达到预设值，或者随机行走算法的收敛条件是否满足。若是，则结束程序；若否，则跳转到（5）继续执行。

在实现随机行走并行算法时使用了 pthread API；同时，为了减少线程间的通信，本论文中使用了 C++ boost 库中的 Random 库[14]，为每一个线程分配了一个独立的随机数生成器，可以很好的保证并行计算的效率。

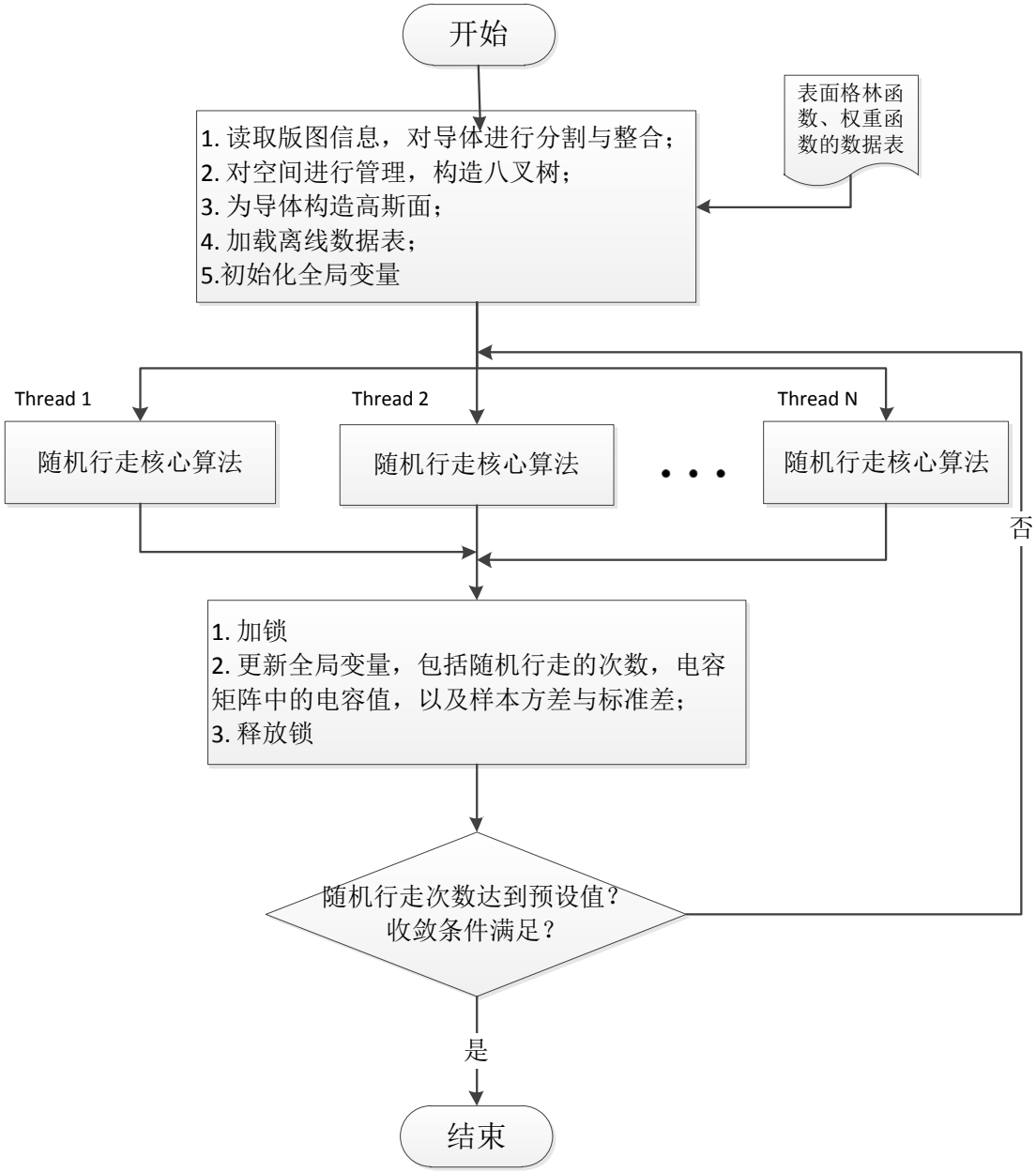


图 4.13 随机行走并行算法示意图

第五章 实验结果

本章将给出空间管理技术的加速方法以及随机行走并行算法的实验结果。测试采用的处理器型号是 48 Intel(R) Xeon(R) CPU E7540 @2.00GH。使用的版图文件是与非门电路。

1、验证随机行走算法得到的电容均值是否符合高斯分布。

本次测试中进行 10000 次抽样，每次抽样的过程是：进行 10000 次随机行走（10^4 个 PATH），得到导体的自容的均值$\overline{C_{self}}$作为一个样本。根据(2.2.17)式，可以计算得到的 10000 次 PATH 所对应的$\overline{C_{self}}$的标准差的理论值近似为 0.04790。统计$\overline{C_{self}}$的样本分布，做出直方图。

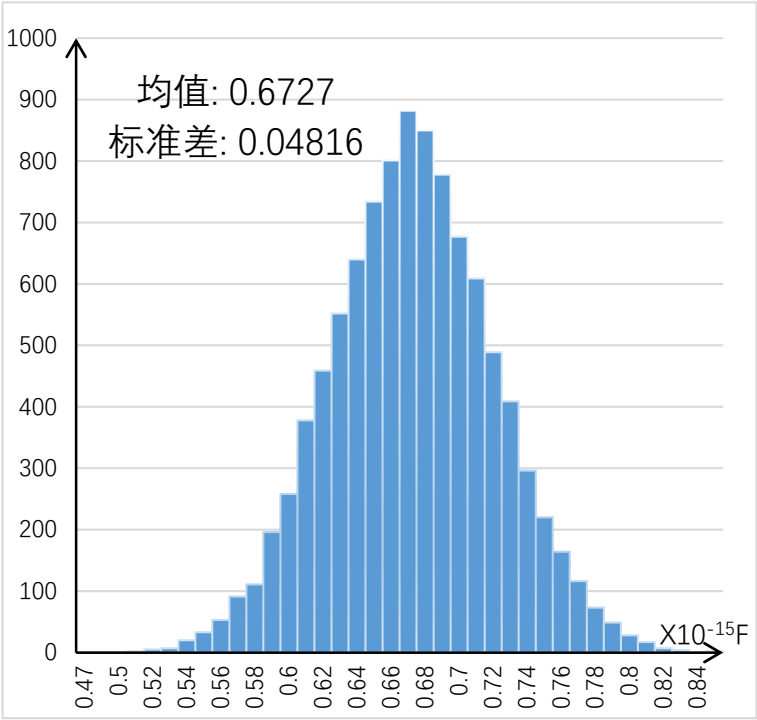


图 5.1 自容的均值的分布图

如图 5.1 所示，$\overline{C_{self}}$的样本均值是$0.6727 \times 10^{-15}F$，$\overline{C_{self}}$的样本标准差为 0.04816，与计算得到的理论值 0.04790 非常接近。由此可以得出结论：自容的均值$\overline{C_{self}}$近似服从高斯分布。

2、随机行走算法提取的电容均值的误差与随机行走(PATH)的次数的关系。

在本次测试中，根据(2.2.18)式计算导体自容的均值的误差$err_{\overline{C_{self}}}$。做出$err_{\overline{C_{self}}}$随 PATH 数目$N_{PATH}$的变化曲线。

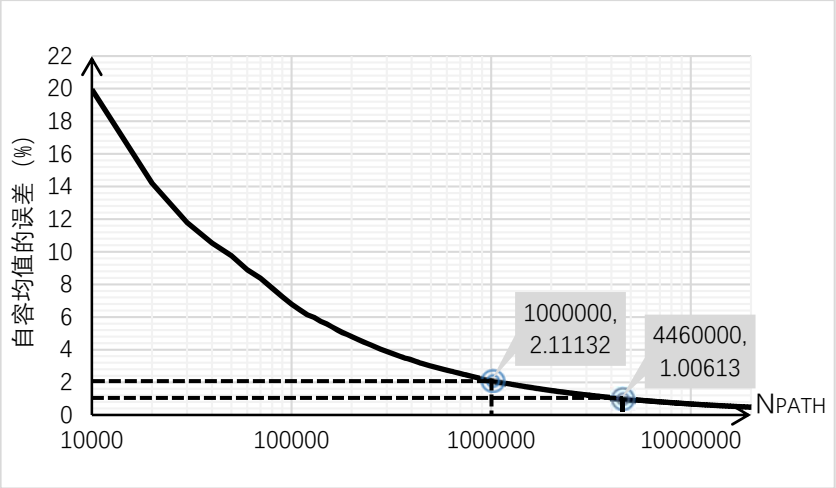


图 5.2 电容均值的误差与随机行走(PATH)数目之间的关系

如图 5.2 所示，随机行走的初始阶段，导体的自容均值的误差$err_{\overline{C_{self}}}$随这随机行走（PATH）的次数$N_{PATH}$增加迅速减小。当$N_{PATH} = 10^6$时，$err_{\overline{C_{self}}}$降至 2%；当$N_{PATH} = 4.46 \times 10^6$时，$err_{\overline{C_{self}}}$降至约 1%。例如可以设置 PATH 数为$10^7$，可以确保随机行走计算得到的自容均值的误差在$\pm$1%以内。

3、不同 k 值条件下生成八叉树的时间T_G以及随机行走一千万次的时间T_{FRW}。

根据 3.4.1 节的介绍，在生成八叉树的过程中，将子节点空间单元的最大跳转距离$L_{子节点}$的初始值$d_{ext子节点}$设置为当前父节点空间单元的最大跳转距离$L_{父节点}$乘以一个缩放因子k。

$$d_{ext子节点} = k \times L_{父节点}$$

这里测试不同 k 值条件下生成八叉树的时间$\boldsymbol{T_G}$以及随机行走一千万次的时间$\boldsymbol{T_{FRW}}$。本次测试中八叉树的最大深度设置为 4，空间单元的候选导体列表的最大长度设置为 4，八叉树根节点的尺寸（即三维数组栅格的尺寸 gridCellSize）设置为 320nm。

表 5.1 生成八叉树加速方法测试结果（10^7 次 PATH）

k	0.4	0.5	0.6	0.7	0.8	0.9	1.0	$d_{ext} = C$
T_G(ms)	187	225	251	315	486	953	2666	2747
T_{FRW}(s)	206	203	190	184	183	184	188	187

[注]：最后一列$d_{ext} = C$指的是每个空间节点的膨胀距离设置为一个固定常数C（C=320nm）

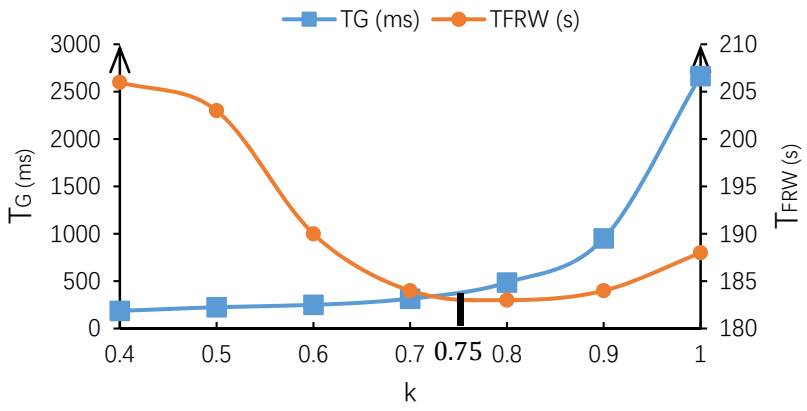


图5.3 k值与T_G、T_{FRW}关系图

通过表 5.1 可以看出，k值越小，生成八叉树的时间越短；另外还可以看出，适当减小k的值，随机行走的速度与d_{ext} = 固定值（$320nm$）的情况相比不下降，原因分析见 3.4.1 节。因此可以说明本文提出的这种加速生成八叉树的方法是可行的。为了在保证随机行走速度的同时减小生成八叉树的时间，可以将k的值设置在 0.7~0.8 左右。

4、检查下一个跳转点是否仍在当前的空间单元对定位空间节点的加速效果。

在本次测试中，八叉树的根节点尺寸固定为 640nm。设定空间单元的候选导体列表的最大长度为 1，这样做的目的是使八叉树尽可能的划分到最大深度。通过调节八叉树的最大深度来间接调节八叉树叶节点的最小尺寸。

表 5.2 候选导体列表方法对定位空间节点的加速效果(10^7 次 PATH)

Depth	hit rate (%)	T^0_{locate} (s)	T^s_{locate} (s)	$T_{speedup}$ (s)	$P_{speedup}$ (%)
1	69.98	17.19	14.59	2.60	15.13
2	58.98	23.56	19.36	4.20	17.83
3	43.50	36.62	33.02	3.60	9.83
4	29.66	60.93	58.69	2.24	3.68
5	21.72	112.35	111.11	1.24	1.10

[注]符号说明：

Depth： 八叉树的最大深度；

hit rate：命中率；

T^0_{locate}： 不检查下一个跳转点是否仍在当前的空间单元时 10^7 次随机行走（PATH）中定位空间节点单元所用的时间；

T_{locate}^{s}：检查下一个跳转点是否仍在当前的空间单元时 10^7 次随机行走（PATH）中定位空间节点单元所用的时间；

$T_{speedup}$：检查下一个跳转点是否仍在当前的空间单元带来的加速时间（10^7 次 PATH）

$$T_{speedup} = T_{locate}^{0} - T_{locate}^{s}$$

$P_{speedup}$：检查下一个跳转点是否仍在当前的空间单元带来的加速百分比（%）

$$P_{speedup} = T_{speedup} \div T_{locate}^{0} \times 100\%$$

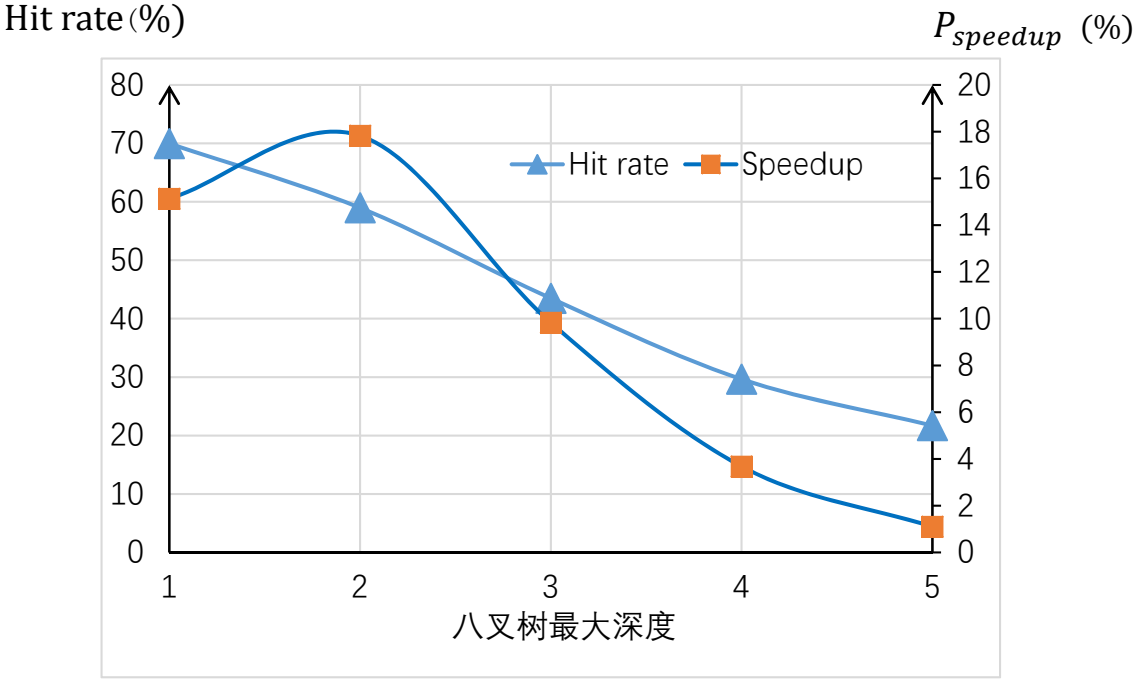


图 5.4 八叉树最大深度与命中率、加速比的关系

从表 5.2 可以得出以下结论。

（1） 八叉树的最大深度越大，命中率越低。这是因为八叉树深度越大，叶子节点空间单元的平均尺寸下降，此时命中率也会降低。

（2） 命中率越低，检查下一个跳转点是否仍在当前的空间单元带来的加速比越小，加速效果越不明显。根据(3.4.3)式可知，当命中率下降时，加速比也将随之下降。

5、为八叉树建立索引方法对定位空间节点的加速效果。

在本次测试中，八叉树的根节点尺寸固定为 640nm，设定空间单元的候选导体列表的最大长度为 1，同时设定八叉树索引数组对应的栅格尺寸始终等于八叉树叶子节点的最小尺寸，目的是将八叉树每一个叶子节点的指针都放进八叉树索引数组中，从而完全避免从八叉树某个父节点开始逐层向下定位叶节点。记录八叉树的最大深度与一千万次随机行走（PATH）中定位叶子节点的时间。

表 5.3 为八叉树建立索引方法对定位空间节点的加速效果(10^7 次 PATH)

Depth	T^0_{locate} (s)	T^s_{locate} (s)	$T_{speedup}$ (s)	$P_{speedup}$ (%)
1	17.27	17.86	-0.59	-3.42
2	23.46	18.79	4.67	19.91
3	36.31	22.09	14.22	39.16
4	58.45	28.88	29.57	50.59
5	95.24	43.01	52.23	54.84
6	216.85	61.98	154.87	71.42

[注]符号说明：

Depth： 八叉树的最大深度；

T^0_{locate}： 不采用为八叉树建立索引方法时 10^7 次随机行走（PATH）中定位空间节点单元所用的时间；

T^s_{locate}： 采用了为八叉树建立索引方法时 10^7 次随机行走（PATH）中定位空间节点单元所用的时间；

$T_{speedup}$：为八叉树建立索引方法带来的加速时间（10^7 次 PATH）

$$T_{speedup} = T^0_{locate} - T^s_{locate}$$

$P_{speedup}$：为八叉树建立索引方法带来的加速百分比（%）

$$P_{speedup} = T_{speedup} \div T^0_{locate} \times 100\%$$

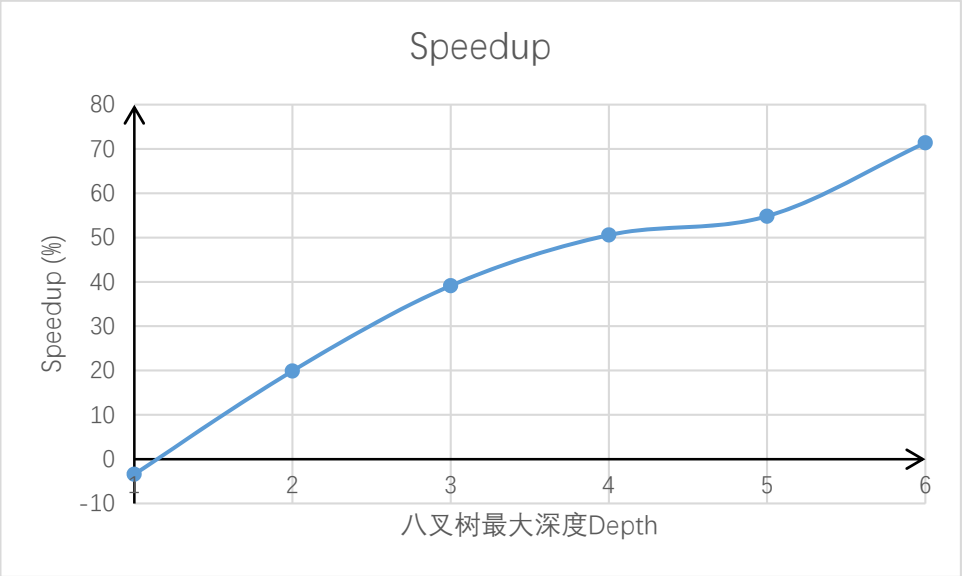


图 5.5 八叉树最大深度与八叉树索引数组加速百分比的关系图

通过表 5.3 可以得出以下结论：

（1） 随着八叉树最大深度的提高，八叉树索引数组方法的加速百分比也越来越大因为随着八叉树的深度D的增加，从八叉树根节点出发定位叶子节点的时间迅速增加；而随着三维数组规模的增加，访问三维数组的时间增加的速度较慢。

（2） 当八叉树深度只有 1 层时，为八叉树建立索引方法没有加速效果。因为八叉树的深度很小，因此可以快速定位叶节点，但实际情况中八叉树深度过小会导致叶子节点的候选导体列表的长度过长，随机行走的速度会变慢，因此一般情况下八叉树的深度会大于 1。

（3） 作为基于八叉树结构的空间分割方法的一种辅助技术，为八叉树建立索引的方法具有明显的加速效果。

6、随机行走过程中的时间分配

在本次测试中八叉树的最大深度设置为 4，空间单元的候选导体列表的最大长度设置为 4，八叉树根节点的尺寸（即三维数组栅格的尺寸 gridCellSize）设置为 320nm。统计进行10^7次随机行走(PATH)时整个算法执行时间的分配情况。

表 5.4 单线程执行10^7次 PATH 时整个程序执行时间的分配情况

步骤	耗时（ms）	占比（%）
读取版图&导体的分割与整合	2	0.00
生成八叉树	2500	1.26
生成高斯面	3	0.00
加载存放格林函数的离线数据表	14189	7.18
10^7 次 PATH 顺序执行的时间	181000	91.56

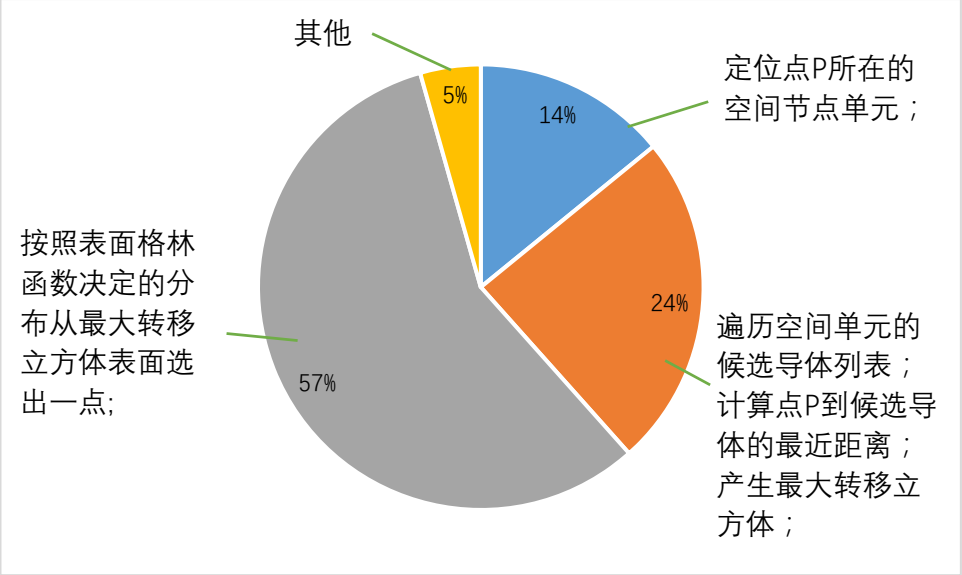


图 5.6 执行一次 PATH 时各个步骤的时间占比(%)

7、随机行走并行算法加速效果

在本次测试中八叉树的根节点的尺寸设置为 320nm，实现了 4.3 节介绍的并行算法。测试在不同线程数目N_{Thread}条件下执行10^7次 PATH 的所需要的时间T_{PATH}。

表 5.4 随机行走时间与线程数目的关系

N_{Thread}	1	2	3	4	5	6	7	8
$T_{PATH}(s)$	173	88	60	44	35	30	27	24
$Speedup$	1.00	1.97	2.88	3.93	4.94	5.77	6.41	7.21

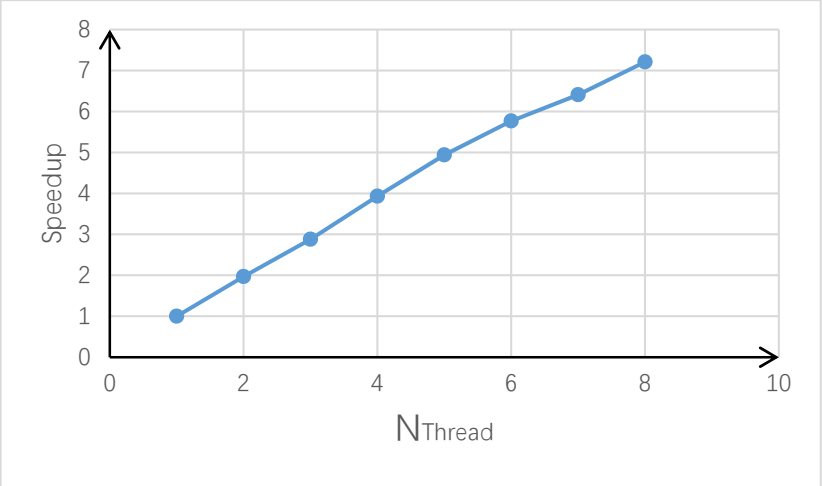


图 5.7 线程数与加速比的关系图

通过图 5.5 可以看出，在使用 8 个线程进行并行计算时可以获得大约7 ×的加速比。由此可见，随机行走并行算法的加速效果较为理想。

第六章 总结与展望

本论文介绍了采用随机行走算法提取三维互连电容的研究背景与国内外的研究现状，利用数学公式详细推导了随机行走算法的基本原理，分析了利用随机行走算法提取大规模集成电路互连电容的优势。本文的核心部分介绍了随机行走算法的具体实现流程，对互连线导体的分割与整合、高斯面的生成、表面格林函数及权重函数的计算、拒绝性采样、并行算法的实现等问题都进行了详细的讨论，并且给出了关键的数据结构。同时，本文重点讨论了空间管理技术在随机行走算法中的重要作用，介绍了与空间管理技术相关的重要概念，实现了基于八叉树结构的空间分割方法，并给出了关键的数据结构及功能描述。此外，本文提出了候选节点单元法以及八叉树索引数组方法等改进技术，有效提高了随机行走过程中定位叶子节点单元的速度。

然而，由于时间有限，本论文对于随机行走算法提取互连电容中的诸多问题没有讨论。例如，如何在多介质层中应用随机行走算法进行电容提取，如何减小随机行走算法的方差使结果更快的收敛，能否将随机行走算法与有限元法、边界元法等算法进行结合以提高电容提取的精度，等等。对于这些问题将在未来的研究工作中进一步探讨。

参考文献

[1] K. Nabors and J. White, “FastCap: A multipole accelerated 3-D capacitance extraction program,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 10, no. 11, pp. 1447–1459, Nov. 1991

[2] N. P. van der Meijs and A. J. van Genderen, “An efficient finite element method for submicron IC capacitance extraction,” in Proc. DAC, Jun. 1989, pp. 678–681.

[3] Y. L. Coz and R. B. Iverson, “A stochastic algorithm for high speed capacitance extraction in integrated circuits,” Solid State Electron., vol. 35, no. 7, pp. 1005–1012, Jul. 1992.

[4] Y. Le Coz, H. J. Greub, and R. B. Iverson, “Performance of random walk capacitance extractors for IC interconnects: A numerical study,” Solid-State Electron., vol. 42, no. 4, pp. 581–588, Apr. 1998.

[5] N. Bansal, “Randomized algorithms for capacitance estimation,” Indian Institute of Technol. Bombay, Mumbai, India, Tech. Rep., Apr. 1999.

[6] R. B. Iverson and Y. Le Coz, “A floating random-walk algorithm for extracting electrical capacitance,” Math. Comput. Simul., vol. 55, pp. 59–66, 2001.

[7] S. H. Batterywala and M. P. Desai, “Variance reduction in Monte Carlo capacitance extraction,” in Proc. 18th Int. Conf. VLSI Design, Jan. 2005, pp. 85–90.

[8] Wenjian Yu, “RWCap: A Floating Random Walk Solver for 3-D Capacitance Extraction of Very-Large-Scale Integration Interconnects” IEEE, vol. 32, no. 3, March 2013.

[9] Chao Zhang and Wenjian Yu, “Efficient Space Management Techniques for Large-Scale Interconnect Capacitance Extraction With Floating Random Walks”, IEEE, vol. 32, no. 10, October 2013.

[10] Zhang C, Yu W. Efficient techniques for the capacitance extraction of chip-scale VLSI interconnects using floating random walk algorithm[C]// Asia and South Pacific Design Automation Conference. 2014:756-761.

[11] Y.-C. Hsiao. (2014). Project CAPLET: Parallelized Capacitance Extraction Toolkit. [Online]. Available: http://www.rle.mit.edu/ cpg/codes/caplet/

[12] Yu-Chung Hsiao, “CAPLET: A Highly Parallelized Field Solver for Capacitance Extraction Using Instantiable Basis Functions”, IEEE, vol. 35, no. 3, March 2016.

[13] Hastings, W. K. (1970-04-01). "Monte Carlo sampling methods using Markov chains and their applications". Biometrika 57 (1): 97–109.

[14] Demming, Robert & Duffy, Daniel J. (2010). Introduction to the Boost C++ Libraries. Volume 1 - Foundations. Datasim. ISBN 978-94-91028-01-4.

致谢

在论文即将完成之际，谨向所有为我提供过指导与帮助的老师和同学们致以诚挚的谢意。

首先需要感谢我的导师严昌浩副教授。我的毕业设计是在严老师的指导下进行的，严老师十分关心我毕业设计的进展情况，对我在设计中遇到的问题给予了悉心指导，并为我提供良好的学习环境。在完成本论文的过程中，严老师对于论文中的关键问题给出了指导性意见，给予了我莫大的鼓励和帮助。我为严老师严谨治学的精神所折服，在此向严老师表达最深的谢意！

另外，我还要感谢在编写程序、搭建开发环境等方面为我提供过帮助的同学和老师们，你们的工作使我的这篇论文得以顺利完成，谢谢！