摘要

从超立方体到环的嵌入在很长一段时间里都是一个开问题。Ching-Jung Guu 在她的博士学位论文中宣称从超立方体到环嵌入的线长问题已经得到了解决，她通过创建派生网络这一概念将该问题转化为了边等周问题，而后者则已经在 L. H. Harper 的论文中得到解决。Guu 依据此得出了格雷码嵌入有最小的环形线长这一结论。然而，Jason Erbele 等人指出了她的证明过程中的错误，这导致 Guu 的结论可能不成立。

在本文中，我们发现了边等周问题的证明过程中的漏洞，并提出了补充证明。我们还开发了一个分布式计算程序——我们称之为 Hyperspark——用来探索较低维度下从超立方体到环的嵌入的所有可能性。我们用我们程序的运行结果验证了 3 阶和 4 阶情况下 Guu 的结论的正确性，并且发现了最优嵌入的一些新特性。这些新特性或许能够为该问题提供一些线索，甚至能够启发整个问题的解决。

关键词： 超立方体；环嵌入；线长问题；分布式计算

Abstract

Embedding of hypercubes into circles has been an open problem for a long time. It was claimed by Ching-Jung Guu in her Ph.D. dissertation that the circular wirelength problem of hypercubes was solved by creating a Derived Network that translates the problem into an Edge Isoperimetric Problem, which was solved by L. H. Harper, based on which she drew a conclusion that the Graycode embedding minimizes the circular wirelength. However, errors in her proof have been spotted by Jason Erbele et al., which may leave Guu's conclusion untenable.

In this thesis, we spot a flaw in the proof of Edge Isoperimetric Problem, and propose a supplementary proof. We also develop a distributed computing program called Hyperspark to explore the possibilities of all embeddings of hypercubes into circles under lower dimensions. Using the results of our program, we have verified the correctness of Guu's conclusion in the cases of 3 and 4 dimensions, and have found out some new traits of optimal embeddings, which may provide clues to the problem and even inspire the entire solution.

Keywords: Hypercube; Circular embedding; Wirelength problem; Distributed computing

目录

插图目录

算法目录

第 1 章 引言

图嵌入问题在计算机科学中有着重要的应用。一方面，现代高性能并行计算机中的任务映射可以被建模为图嵌入问题：首先它将任务作为节点，将任务间的通讯作为边，就能得到一个任务图；接下来它将映射模拟为从一个图到另一个图的嵌入，并试图寻找这个嵌入的最小线长。另一方面，多处理器的排列和连线也能转化为图嵌入问题。一般意义上，图嵌入问题都是 NP 完全的 [16]，我们只能利用启发式方法来探究一些特殊的图结构用以应用到这些研究中，其中超立方体是一种重要且高效的并行结构，但是其缺点是连线复杂；而链、环、圆柱和网格等结构都具有连线简单的特点。尽管有很多常规的图嵌入问题已经得以解决了，如从超立方体到链 [7]、从超立方体到网格 [4]、从二叉树到网格 [6] 等，然而从超立方体到环的嵌入则依然是一个开问题，从超立方体到圆柱的嵌入则依赖于到环的研究 [10]。

分布式计算则是计算机科学中的另一热门领域。分布式计算技术为研发高性能计算机开辟了另一条路径：普通计算机通过网络连接之后，可以利用这种技术成为分布式计算集群中的一个节点，用以完成先前只能在传统意义上的高性能计算机（如超级计算机）上进行的工作。随着科学技术的发展和进步，分布式计算已经使参与其中的所有普通计算机的“联合计算能力”超过了单台超级计算机。分布式计算虽然已经有几十年的研究历史，但现在依然是计算机研究领域中的一片热土，并且已被广泛地运用于各种需要高强度计算能力的应用中，如科学计算、海量数据分析处理等。

本文将对从超立方体到环的嵌入性质进行研究。首先，我们在第 2 章对本文所用到的术语和背景知识做简单的介绍。接下来在第 3 章，我们简要地概括和说明现有的从超立方体到链和环的嵌入研究，并给出对其中一部分结果的改进。第 4、5 章是本文的主要内容：我们在第 4 章介绍分布式计算程序 Hyperspark，利用其对 3 阶和 4 阶超立方体进行嵌入和线长的计算，并在第 5 章对计算结果做简单的分析。第 6 章则是对本文的总结，以及对未来在该研究领域进一步工作的展望。

第 2 章　背景知识

2.1　图

一个图（Graph）$G = (V_G, E_G, \partial_G)$ 是一个三元组，其中 V_G 是顶点的集合，E_G 是边的集合，$\partial_G \colon E_G \to \binom{V_G}{1} \cup \binom{V_G}{2}$ 则是从边集到顶点集的可重复二元子集的一个映射函数。∂_G 指出了一条边对应了哪两个顶点。下面我们列出了三种在本文中会用到的图结构。

d 阶超立方体（Hypercube），我们用 Q_d 来表示。$Q_d = (V_{Q_d}, E_{Q_d}, \partial_{Q_d})$，其中 $V_{Q_d} = \{0,1\}^d$（即 $\{0,1\}$ 的 d 倍笛卡尔积，$|V_{Q_d}| = 2^d$）。对于 $e \in E_{Q_d}$ 和 $v, w \in V_{Q_d}$，$\partial_{Q_d}(e) = \{v, w\}$ 当且仅当 v 和 w 有正好一位坐标不相同。Q_3 的示意图见图 2-1。

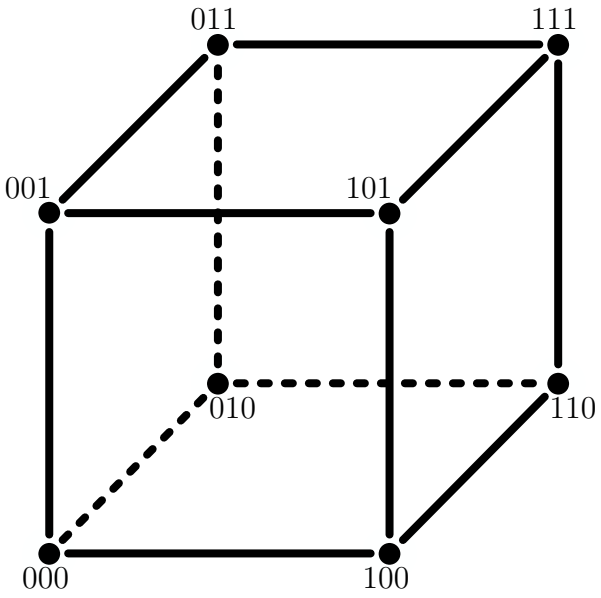


图 2-1　Q_3 的示意图

长为 n 的链，也称为路径（Path），我们用 P_n 来表示。$P_n = (V_{P_n}, E_{P_n}, \partial_{P_n})$，其中 $V_{P_n} = \{0, 1, 2, \ldots, n-1\}$。对于 $e \in E_{P_n}$ 和 $i, j \in V_{P_n}$，$\partial_{C_n}(e) = \{i, j\}$ 当且仅当 $j = i + 1$。P_4 的示意图见图 2-2。

图 2-2　P_4 的示意图

长为 n 的环（Circle），我们用 C_n 来表示。$C_n = (V_{C_n}, E_{C_n}, \partial_{C_n})$，其中 $V_{C_n} = V_{P_n}$。对于 $e \in E_{C_n}$ 和 $i, j \in V_{C_n}$，$\partial_{C_n}(e) = \{i, j\}$ 当且仅当 $j \equiv (i+1) \bmod n$。C_4 的示意图见图 2-3。

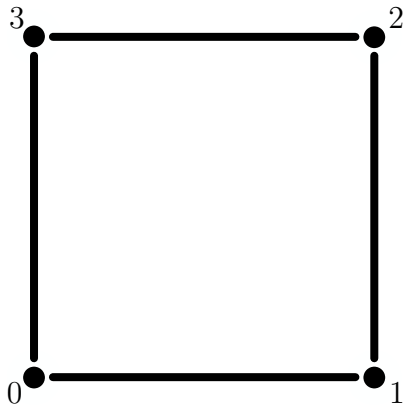


图 2-3　C_4 的示意图

给定图 $G = (V_G, E_G, \partial_G)$ 和 $v, w \in V_G$，一条从 v 到 w 且长度为 n 的路径 $P(v, w)$ 是一个导出自 G 的长为 n 的链，其中 v 可视为 0，w 可视为 $n-1$，$|P(v, w)| = n$。从 v 到 w 的距离（Distance）$d_G(v, w)$ 是从 v 到 w 的最短路径的长度。

2.2　线长问题

有了上述关于图的基本概念的定义，我们便可以来定义线长等一系列概念了。

给定图 $G = (V_G, E_G, \partial_G)$、图 $H = (V_H, E_H, \partial_H)$ 和单射函数 $\eta\colon V_G \to V_H$，其中 $|V_G| \le |V_H|$，那么在 η 下从 G 到 H 的线长（Wirelength）为

$$wl(G, H, \eta) = \sum_{\substack{e \in E_G \\ \partial_G(e) = \{v, w\}}} d_H(\eta(v), \eta(w)) \tag{2-1}$$

即 G 中所有的边在 η 下嵌入到 H 后的距离之和。从 G 到 H 的线长为

$$wl(G, H) = \min_{\eta} wl(G, H, \eta) \tag{2-2}$$

从 G 到 H 的线长问题（Wirelength Problem）即寻找 $wl(G, H)$ 和能达到最小线长的单射函数 η。

本文所研究的问题为从超立方体到环嵌入的线长问题，即对 $\forall d > 2$，寻找 $wl(Q_d, C_{2^d})$ 和能达到最小线长的双射函数 η。注意 η 在目标图为链或环时也被称为*编号方式*（Numbering），因为链和环的顶点集都是由自然数构成。

在下文中，若无特殊说明，$n = 2^d$，V、E、∂ 等符号都特指超立方体的相应元素。大多数情况下，读者应该都能够根据上下文判断出符号的意义。

2.3 格雷码编号方式

格雷码（Graycode）在计算机组成原理、体系结构以及通信领域中有着重要的作用，并且研究人员已证明（或相信）格雷码编号方式是从超立方体到链、环、圆柱等结构的嵌入的线长问题的解 [7,3,5]。

对于从超立方体到环的嵌入，格雷码编号方式 $\mathcal{G}\colon V_{Q_d} \to V_{C_{2^d}}$ 有如下的递归定义：

(1) 首先我们有 Q_1 上的格雷码编号方式 $\mathcal{G}\colon V_{Q_1} \to V_{C_2}$ 为

$$\mathcal{G}(0) = 0, \qquad \mathcal{G}(1) = 1 \tag{2-3}$$

(2) 如果给定了 Q_d 上的格雷码编号方式 $\mathcal{G}\colon V_{Q_d} \to V_{C_{2^d}}$（$d \geq 1$），那么我们给 Q_d 的每一个顶点添加一个坐标前缀 0 和一个相邻的顶点。这些新的顶点则给出了一个 Q_d 的拷贝，我们为其添加一个坐标前缀 1。这个新的 Q_d 的编号与原先的相反，即与编号为 $2^d - 1$ 的顶点相邻的顶点编号为 2^d、与编号为 $2^d - 2$ 的顶点相邻的顶点编号为 $2^d + 1$、……、与编号为 0 的顶点相邻的顶点编号为 $2^{d+1} - 1$。这样我们便得到了 Q_{d+1} 上的格雷码编号方式了。

值得注意的是，格雷码编号方式有一个显著的特点：C_{2^d} 中任意两个相邻的顶点对应在 Q_d 中都有边相连。

2.4 立方集

一个 d 阶立方的 c 阶*子立方*（Subcube）是 Q_d 的子图，导出自一个包含所有在 d–c 个坐标下有相同（固定）值的顶点的集合。例如，图 2-1 所展示的 Q_3 有 8 个 2 阶子立方（对应立方体有 8 个面）和 12 个 1 阶子立方（对应立方体有 12 条边）。

一个 d 阶立方的 c 阶子立方的*邻居*（Neighbor）是在 d–c 个固定坐标下有正好一个坐标不同的任意 c 阶子立方。我们再以图 2-1 所展示的 Q_3 为例，对于

Q_3 中的每一个 2 阶子立方，它的邻居是可以与它一起构成 Q_3 的任意 2 阶子立方，因此 Q_3 中的每一个 2 阶子立方都只有 1 个邻居；而对于 Q_3 中的每一个 1 阶子立方，它的邻居是可以与它一起构成一个 2 阶子立方的任意 1 阶子立方，因此 Q_3 中的每一个 1 阶子立方都有 2 个邻居。

Q_d 中如果 $S \subseteq V$ 是一个互不相交的不同阶子立方的并集，并且其中每个 c_i 阶子立方都满足：对于其他所有 c_j 阶子立方（$c_j > c_i$），该 c_i 阶子立方处在该 c_j 阶子立方的一个邻居中，那么 S 就被称为**立方集**（Cubal Set）或**复合集**（Composite Set）。关于立方集更详细的介绍，请参考 [2]。

第 3 章 历史研究

3.1 边等周问题

在图论中，大量的问题都可以被转化为边等周问题并得到解决，包括一些图嵌入问题。在 L. H. Harper 的著作 [7,2] 中，他通过一系列的转化，利用边等周问题证明了从超立方体到链嵌入的线长问题。

下面我们给出边等周问题的定义，再简要概述如何利用其解决从超立方体到链嵌入的线长问题，并给出 Q_d 上的边等周问题证明过程的改进。注意这种转化方式为我们提供了一种重要的问题解决思路。

3.1.1 定义

对于任意的图 G 和 $S \subseteq V_G$，令

$$\Theta(S) = \{e \in E_G \colon \partial_G(e) = \{v, w\}, v \in S, w \notin S\} \tag{3-1}$$

并称之为 S 的边界（Edge-boundary）。那么对于一个给定的图 G 和自然数 k，边等周问题（Edge-Isoperimetric Problem, EIP）是对于所有的 $S \subseteq V_G$ 并且 $|S| = k$，寻找

$$\min_{|S|=k} |\Theta(S)| \tag{3-2}$$

和能达到该最小值的子集 S。

为了便于下文叙述，我们在这里定义一个与 EIP 类似的问题，并给出其在 Q_d 下与 EIP 的关系。

对于任意的图 G 和 $S \subseteq V_G$，令

$$E(S) = \{e \in E_G \colon \partial_G(e) = \{v, w\}, v \in S, w \in S\} \tag{3-3}$$

并称之为 S 的导出边集（Induced Edges）。对于一个给定的图 G 和自然数 k，导出边问题（Induced Edge Problem）是对于所有的 $S \subseteq V_G$ 并且 $|S| = k$，寻找

$$\max_{|S|=k} |E(S)| \tag{3-4}$$

和能达到该最大值的子集 S。

引理 3.1. 如果图 $G=(V_G,E_G,\partial_G)$ 是一个 δ 度正则图，那么对于 $\forall S\subseteq V_G$，都有

$$|\Theta(S)|+2|E(S)|=\delta|S| \tag{3-5}$$

证明. $\delta|S|$ 表示 S 对应的边数，然而出现在 $E(S)$ 中的边被计算了两次。 □

根据引理 3.1，我们可以知道 Q_d 上的导出边问题与 EIP 等价，因为对于 $\forall S\subseteq V$ 和 $\forall k$，$\min_{|S|=k}|\Theta(S)|=kd-2\max_{|S|=k}|E(S)|$。

3.1.2 线长问题到 EIP

回想一下，对于给定的编号方式 $\eta\colon V_{Q_d}\to V_{P_n}$，在 η 下从 Q_d 到 P_n 的线长 $wl(Q_d,P_n,\eta)$ 为

$$wl(Q_d,P_n,\eta)=\sum_{\substack{e\in E\\ \partial(e)=v,w}} d_{P_n}(\eta(v),\eta(w))=\sum_{\substack{e\in E\\ \partial(e)=v,w}}|\eta(v)-\eta(w)| \tag{3-6}$$

而从 Q_d 到 P_n 的线长问题则是寻找

$$wl(Q_d,P_n)=\min_{\eta} wl(Q_d,P_n,\eta) \tag{3-7}$$

为了将从 Q_d 到 P_n 的线长问题转化为 Q_d 上的 EIP，首先我们令

$$S_k(\eta)=\eta^{-1}(\{0,1,\dots,k-1\})=\{v\in V\colon \eta(v)\le k\} \tag{3-8}$$

即编号方式 η 下的前 k 个顶点的集合。接下来，我们令

$$\chi(e,k)=\begin{cases}1 & \text{如果 } \partial(e)=\{v,w\},\eta(v)\le k<\eta(w)\\ 0 & \text{其他情况}\end{cases} \tag{3-9}$$

那么我们就有

$$wl(Q_d,P_n,\eta)=\sum_{\substack{e\in E\\ \partial(e)=\{v,w\}}}|\eta(v)-\eta(w)|=\sum_{e\in E}\sum_{k=0}^{n}\chi(e,k) \tag{3-10}$$

$$=\sum_{k=0}^{n}\sum_{e\in E}\chi(e,k)=\sum_{k=0}^{n}|\Theta(S_k(\eta))| \tag{3-11}$$

因此，

$$wl(Q_d, P_n) = \min_{\eta} wl(Q_d, P_n, \eta) \tag{3-12}$$

$$= \min_{\eta} \sum_{k=0}^{n} |\Theta(S_k(\eta))| \tag{3-13}$$

$$\geq \sum_{k=0}^{n} \min_{\eta} |\Theta(S_k(\eta))| = \sum_{k=0}^{n} \min_{|S|=k} |\Theta(S)| \tag{3-14}$$

根据该式，我们可以得到一个结论。

定理 3.1. *对于 $\forall \eta: V_{Q_d} \to V_{P_n}$，如果它的所有初始段 $S_k(\eta)$（$0 \leq k \leq n$）都是 EIP 的解，那么它本身就是从 Q_d 到 P_n 的线长问题的一个解。*

根据定理 3.1，我们将问题转化为了 EIP。

3.1.3 EIP 证明的改进

对于 Q_d 上的 EIP，L. H. Harper 在一篇论文 [7] 中提出了下面这个定理，并给出了他的证明。

定理 3.2. *$S \subseteq V_{Q_d}$ 在基数为 k 时有最大的 $|E(S)|$，当且仅当 S 是立方集。*

在定理 3.2 的证明过程中，Harper 对 d 进行数学归纳，并用到了下面这个性质：如果 $2^{d-1} \leq k < 2^d$，那么

$$E(k+1) - E(k) = E(k - 2^{d-1} + 1) - E(k - 2^{d-1}) + 1 \tag{3-15}$$

其中 $E(k)$ 表示一个 k-立方集 S 的导出边数 $|E(S)|$，因为 $|E(S)|$ 并非由 d 决定，而仅仅由 $k = |S|$ 决定。

在这篇论文发表两年后，A. J. Bernstein 发表了一篇后续论文 [11]，指出 Harper 忽视了一种情况。他在这篇后续论文中提出了下面这个引理，用该引理修补了 Harper 的证明中的漏洞：

引理 3.2. *对于 $\forall d$ 和 $\forall k, t > 0$ 使得 $k + t < 2^d$，都有*

$$E(t) < E(k+t) - E(k) < E(2^d) - E(2^d - t) \tag{3-16}$$

在引理 3.2 的证明过程中，Bernstein 对 d 进行数学归纳，然后分别考虑了三种情况：

(1) 当 $k \geq 2^{d-1}$ 时，左右两边不等式成立；

(2) 当 $k+t \leq 2^{d-1}$ 时，左右两边不等式成立；

(3) 当 $k < 2^{d-1} < k+t$ 时，左右两边不等式成立。

然而我们发现第 3 种情况的证明依然存在纰漏：Bernstein 并未考虑 $t \geq 2^{d-1}$ 的情况，即当 $k < 2^{d-1} < k+t$ 时，

$$E(2^{d-1}) - E(k) > E(t) - E(t-(2^{d-1}-k)) \tag{3-17}$$

仅对 $t < 2^{d-1}$ 成立。下面我们给出经过补充的情况 3 的完整证明。

引理 3.2 情况 3 的完整证明. 当 $k < 2^{d-1} < k+t$ 时，

(1) 如果 $t \geq 2^{d-1}$，那么对于左边的不等式，我们有

$$\begin{aligned}
E(k+t) - E(k) &= \left[E(k+t) - E(2^{d-1})\right] + \left[E(2^{d-1}) - E(k)\right] && (3\text{-}18)\\
&= \left[E(k+t-2^{d-1}) + k + t - 2^{d-1}\right] + \left[E(2^{d-1}) - E(k)\right] && (3\text{-}19)\\
&= \left[E(k+t-2^{d-1}) - E(k)\right] + E(2^{d-1}) + k + t - 2^{d-1} && (3\text{-}20)\\
&> E(t-2^{d-1}) + E(2^{d-1}) + t - 2^{d-1} + k && (3\text{-}21)\\
&= E(t) + k > E(t) && (3\text{-}22)
\end{aligned}$$

对于右边的不等式，我们有

$$\begin{aligned}
E(k+t) - E(k) &= \left[E(k+t) - E(2^{d-1})\right] + \left[E(2^{d-1}) - E(k)\right] && (3\text{-}23)\\
&= \left[E(k+t-2^{d-1}) + k + t - 2^{d-1}\right] + \left[E(2^{d-1}) - E(k)\right] && (3\text{-}24)\\
&= \left[E(k+t-2^{d-1}) - E(k)\right] + E(2^{d-1}) + k + t - 2^{d-1} && (3\text{-}25)\\
&< E(2^{d-1}) - E(2^d - t) + E(2^{d-1}) + k + t - 2^{d-1} && (3\text{-}26)\\
&= 2E(2^{d-1}) + k + t - 2^{d-1} - E(2^d - t) && (3\text{-}27)\\
&< 2E(2^{d-1}) + 2^{d-1} - E(2^d - t) && (3\text{-}28)\\
&= E(2^d) - E(2^d - t) && (3\text{-}29)
\end{aligned}$$

(2) 相应地，如果 $t < 2^{d-1}$，那么对于左边的不等式，我们有

$$E(k+t)-E(k) = \big[E(k+t)-E(2^{d-1})\big] + \big[E(2^{d-1}) - E(k)\big] \tag{3-30}$$
$$= \big[E(k+t-2^{d-1}) + k + t - 2^{d-1}\big] + \big[E(2^{d-1}) - E(k)\big] \tag{3-31}$$
$$> E(k+t-2^{d-1}) + \big[E(t)-E(t-(2^{d-1}-k))\big] \tag{3-32}$$
$$= E(t) \tag{3-33}$$

对于右边的不等式，我们有

$$E(k+t) - E(k) = \big[E(k+t) - E(2^{d-1})\big] + \big[E(2^{d-1}) - E(k)\big] \tag{3-34}$$
$$= \big[E(k+t-2^{d-1}) + k + t - 2^{d-1}\big] + \big[E(2^{d-1}) - E(k)\big] \tag{3-35}$$
$$= \big[E(k+t-2^{d-1}) - E(k)\big] + E(2^{d-1}) + k + t - 2^{d-1} \tag{3-36}$$
$$< E(2^{d-1}) - E(2^{d-1} - t) + t + k - 2^{d-1} \tag{3-37}$$
$$= E(2^d) - E(2^d - t) + k - 2^{d-1} \tag{3-38}$$
$$< E(2^d) - E(2^d - t) \tag{3-39}$$

□

3.2 派生网络

从超立方体到链嵌入的线长问题可以通过 EIP 得到解决，那么从超立方体到环的嵌入呢？1997 年，Ching-Jung Guu 在她的博士学位论文 [3] 中宣称从超立方体到环嵌入的线长问题已经得到了解决。她通过创建派生网络这一概念将该问题转化为了 EIP，并得出了格雷码嵌入有最小的环形线长这一结论。

下面我们给出派生网络及其一系列附加概念的定义，再简要概述如何利用派生网络将从超立方体到环嵌入的线长问题转化为 EIP，最后我们简述 Guu 的证明思路，并援引 Jason Erbele 等人的文章 [8]，这篇文章指出了 Guu 的证明过程中的错误。

3.2.1 定义

一个派生网络（Derived Network）是一个定义在 Q_d 上的顶点加权图 $D = (V_D, E_D, \partial_D)$，其中 V_D 由 V_{Q_d} 的所有基数为 2^{d-1} 的子集构成。对于 $e \in E_D$ 和 $V, W \in V_D$，$\partial_D(e) = \{V, W\}$ 当且仅当 $V \Delta W = 2$，即 $|V \cap W| = 2^{d-1} - 1$。对于 D 中的顶点 $V \in V_D$，它的权重为 $|\Theta(V)|$。

一条派生网络中的路径 $\mathcal{P}=(\mathcal{P}_1,\mathcal{P}_2,\ldots,\mathcal{P}_{2^{d-1}},\mathcal{P}_{2^{d-1}+1})$ 是 D 中的一条长为 2^{d-1} 的路径，我们用 $2^{d-1}+1$ 个 V_D 中的顶点（即 $\mathcal{P}_i$）来表示，其中相邻两个顶点之间都有边相连。Guu 将 $\mathcal{P}$ 设计为对应一个编号方式 $\eta_{\mathcal{P}}\colon V_{Q_d}\to V_{C_n}$，其中

$$\eta_{\mathcal{P}}(\mathcal{P}_1-\mathcal{P}_2)=0,\qquad \eta_{\mathcal{P}}(\mathcal{P}_2-\mathcal{P}_1)=2^{d-1}, \tag{3-40}$$

$$\eta_{\mathcal{P}}(\mathcal{P}_2-\mathcal{P}_3)=1,\qquad \eta_{\mathcal{P}}(\mathcal{P}_3-\mathcal{P}_2)=2^{d-1}+1, \tag{3-41}$$

$$\cdots\qquad\qquad\cdots$$

$$\eta_{\mathcal{P}}(\mathcal{P}_{2^{d-1}-1}-\mathcal{P}_{2^{d-1}})=2^{d-1}-2,\qquad \eta_{\mathcal{P}}(\mathcal{P}_{2^{d-1}}-\mathcal{P}_{2^{d-1}-1})=2^{d}-2, \tag{3-42}$$

$$\eta_{\mathcal{P}}(\mathcal{P}_{2^{d-1}}-\mathcal{P}_{2^{d-1}+1})=2^{d-1}-1,\qquad \eta_{\mathcal{P}}(\mathcal{P}_{2^{d-1}+1}-\mathcal{P}_{2^{d-1}})=2^{d}-1 \tag{3-43}$$

这样一来，我们便有

$$\mathcal{P}_1=\eta_{\mathcal{P}}^{-1}(\{0,1,\ldots,2^{d-1}-1\}), \tag{3-44}$$

$$\mathcal{P}_2=\eta_{\mathcal{P}}^{-1}(\{1,2,\ldots,2^{d-1}\}), \tag{3-45}$$

$$\mathcal{P}_3=\eta_{\mathcal{P}}^{-1}(\{2,3,\ldots,2^{d-1}+1\}), \tag{3-46}$$

$$\cdots$$

$$\mathcal{P}_{2^{d-1}-1}=\eta_{\mathcal{P}}^{-1}(\{2^{d-1}-2,2^{d-1}-1,\ldots,2^{d}-3\}), \tag{3-47}$$

$$\mathcal{P}_{2^{d-1}}=\eta_{\mathcal{P}}^{-1}(\{2^{d-1}-1,2^{d-1},\ldots,2^{d}-2\}), \tag{3-48}$$

$$\mathcal{P}_{2^{d-1}+1}=\eta_{\mathcal{P}}^{-1}(\{2^{d-1},2^{d-1}+1,\ldots,2^{d}-1\}) \tag{3-49}$$

我们注意到 $\mathcal{P}_{2^{d-1}+1}$ 是 $\mathcal{P}_1$ 在 V_{Q_d} 中的补集，因此我们也将 $\mathcal{P}_{2^{d-1}+1}$ 标记为 $\mathcal{P}_1^C$。

给定任意的 $S\subseteq V_{Q_d}$，考虑 Q_d 的所有 $d-1$ 阶子立方，我们一共有 $2d$ 个这样的子立方，并将它们标记为 H_i，那么我们定义 S 的类型（Type）为

$$Type(S)=\min_{1\le i\le 2d}|S\cap H_i| \tag{3-50}$$

路径 $\mathcal{P}$ 的类型序列（Type Sequence）则为

$$T(\mathcal{P})=\big(Type(\mathcal{P}_1),Type(\mathcal{P}_2),\ldots,Type(\mathcal{P}_{2^{d-1}}),Type(\mathcal{P}_1^C)\big) \tag{3-51}$$

注意 $Type(S)\in\{0,1,\ldots,2^{d-2}\}$。Guu 把 $Type(S)$ 在 0 和 2^{d-3} 之间的子集 S 称为小集（Small），把其余的称为大集（Big）。

3.2.2 派生网络到 EIP

首先我们给出一个等式

$$wl(Q_d, C_n, \eta) = \sum_{i=0}^{2^{d-1}-1} |\Theta(\eta^{-1}(\{i, i+1, \ldots, i+(2^{d-1}-1)\}))| \tag{3-52}$$

该等式的证明类似于第 3.1.2 小节中线长问题到 EIP 的转化，只不过目标图从 P_n 变为了 C_n。

根据上一小节中路径 $\mathcal{P}$ 的定义，我们有

$$wl(Q_d, C_n, \eta_{\mathcal{P}}) = \sum_{i=0}^{2^{d-1}-1} |\Theta(\eta_{\mathcal{P}}^{-1}(\{i, i+1, \ldots, i+(2^{d-1}-1)\}))| \tag{3-53}$$

$$= \sum_{i=1}^{2^{d-1}} |\Theta(\mathcal{P}_i)| \tag{3-54}$$

即派生网络中的一条路径所对应的编号方式的线长等于该路径中所有顶点（不包含 $\mathcal{P}_1^C$）的权重之和。因此从超立方体到环嵌入的线长问题就是寻找派生网络中有最小权重之和的路径，从而该问题被转化为了 EIP。

3.2.3 证明与缺陷

通过派生网络把线长问题转化为 EIP 后，Guu 对格雷码编号方式有最小的环形线长这一结论进行了证明。

首先，令 $\mathcal{P}_{\mathcal{G}}$ 表示格雷码编号方式在派生网络中对应的路径，Guu 计算了 $\mathcal{P}_{\mathcal{G}}$ 的类型序列

$$T(\mathcal{P}_{\mathcal{G}}) = (0, 1, \ldots, 2^{d-3}, 2^{d-3}-1, \ldots, 0, 1, \ldots, 2^{d-3}, 2^{d-3}-1, \ldots, 0) \tag{3-55}$$

她发现 $\mathcal{P}_{\mathcal{G}}$ 中每一个顶点都是小集，并且一共有两个类型为 2^{d-3}，三个类型为 0，四个类型为 i（$0 < i < 2^{d-3}$）。

然后，令 $\theta(d, k)$ 表示 Q_d 上 EIP 的解，即

$$\theta(d, k) = \min_{\substack{S \subseteq V_{Q_d} \\ |S|=k}} |\Theta(S)| \tag{3-56}$$

她利用 Harper 的结论推导出：对于任意的 $V \in V_D$ 且 $Type(V) = t \leq 2^{d-3}$（即 V 为小集），

$$|\Theta(V)| \geq 2\theta(d-1, t) + (2^{d-1} - 2t) \tag{3-57}$$

并指出 $\mathcal{P}_{\mathcal{G}}$ 中的每一个顶点都达到了这个下界。

接下来，她对派生网络中的任意路径 $\mathcal{P}$ 构造了一系列变换，并说明只要证明了对于任意的 $V \in V_D$ 且 $Type(V) = t \geq 2^{d-3}$（即 V 为大集），

$$|\Theta(V)| \geq \frac{3}{4} \cdot 2^d \tag{3-58}$$

就能利用这一系列变换证明

$$wl(Q_d, C_n, \eta_{\mathcal{P}}) \geq wl(Q_d, C_n, \mathcal{G}) \tag{3-59}$$

从而证明格雷码编号方式是最优的。

Guu 的论文的余下篇幅都是在证明不等式 (3-58) 的成立。然而，Jason Erbele 等人在 [8] 中指出了她的证明过程中的错误：在她证明该不等式的最后一步，存在 $S_1 \cup S_2 = S$ 和 $S_1 \cap S_2 \neq \emptyset$，她却将 S_1 和 S_2 误以为是 S 的划分，因而利用了 $|S_1| + |S_2| = |S|$ 这一错误的结论用以完成证明。

3.3 边拥塞与划分引理

在图嵌入领域，Paul Manuel 等人提出了另一种用于线长问题的方法，并且他们用该方法成功地解决了从超立方体到网格的嵌入的线长问题 [4]。下面我们就简要介绍他们的研究。

给定图 G、H 和双射函数 $\eta\colon V_G \to V_H$，对于边 $e \in E_H$，我们定义

$$EC_\eta(G, H, e) = \{\{v, w\} \in E_G \colon e \in P(\eta(v), \eta(w))\} \tag{3-60}$$

并称之为 e 的边拥塞（Edge Congestion）。相应地，我们则把 $|EC_\eta(G, H, e)|$ 称为 e 的边拥塞度。而在 η 下从 G 到 H 的边拥塞度为

$$|EC_\eta(G, H)| = \max_{e \in E_H} |EC_\eta(G, H, e)| \tag{3-61}$$

为简洁起见，后文中我们都将用 $EC_\eta(e)$ 来表示 $EC_\eta(G, H, e)$。注意若有 $E \subseteq E_H$，则 $EC_\eta(E) = \bigcup_{e \in E} EC_\eta(e)$。

下面两个引理在 [4] 中已被证明。注意 H 中的一个边集被认为是一个边割集（Edge Cut），当从 H 中去掉这些边时会导致 H 不连通。

引理 3.3 (拥塞引理). 令图 $G = (V_G, E_G, \partial_G)$ 是一个 δ 度正则图，η 是从图 G 到图 H 的一个嵌入。令 S 是 H 的一个边割集，并且去掉 S 会将 H 分割成 H_1 和 H_2 这两个部分。令 $G_1 = \eta^{-1}(H_1)$，$G_2 = \eta^{-1}(H_2)$。另外，S 还需满足下列条件：

(i) 对于 $i \in \{1,2\}$ 和每一条边 $\{v,w\} \in E_{G_i}$，$P(v,w)$ 不在 S 中有边；

(ii) 对于每一条边 $\{v,w\}$ 满足 $v \in G_1$ 和 $w \in G_2$，$P(v,w)$ 在 S 中只有一条边；

(iii) G_1 是最优的，即对于任意 $V \subseteq V_G$ 且 $|V| = |V_{G_1}|$，$|E(V_{G_1})| \leq |E(V)|$。

那么 $|EC_\eta(S)|$ 是最小的，即对于所有从 G 到 H 的嵌入 η'，都有 $|EC_\eta(S)| \leq |EC_{\eta'}(S)|$。

引理 3.4 (划分引理). 令 η 是从 G 到 H 的一个嵌入。若存在 $\{S_1, S_2, \ldots, S_p\}$ 是 E_H 的一个划分，其中每一个 S_i 都是 H 的一个边割集，那么

$$wl(G,H) = wl(G,H,\eta) = \sum_{i=1}^{p} |EC_\eta(S_i)| \tag{3-62}$$

引理 3.4 为图嵌入领域中的线长问题提供了一种新的解决思路。然而它并不适用于从超立方体到环嵌入的线长问题，因为我们找不到 E_{C_n} 的一个这样的划分。

第 4 章 Hyperspark

正如在第 3 章所见，目前我们还没有寻找到一个从理论上解决超立方体到环嵌入的线长问题的方法。但是这并不表示我们对此就束手无策了。一个显而易见的思路是：对于给定的阶 d，编号方式 $\eta\colon V_{Q_d} \to V_{C_n}$ 的个数必然是有限的。既然如此，我们为何不一一穷举所有的编号方式来寻找最小线长呢？然而问题是，由于 Q_d 有 2^d 个顶点，意味着存在 $2^d!$ 个编号方式。当 $d = 3$ 时，$2^d! = 8! = 40,320$ 还是一个较小的数目，可以仅通过单机穷举得到解决。但是当 $d = 4$ 时，$2^d! = 16! = 20,922,789,888,000$ 就是一个较大的计算量了，更不用说 $d > 4$ 时的情况了。因此我们不能寄希望于通过穷举所有的编号方式来解决从超立方体到环嵌入的线长问题。

然而，我们还有另外一种思路：首先利用程序计算出低阶情况（如 $d = 3, 4$）下的所有最优编号方式，再对它们进行数据处理和分析，希望从中寻找出一些通用的特性、规律等用以辅助问题的解决。正如前面所述，当 $d = 4$ 时问题的规模已经大到难以单机解决了。这时我们就需要依赖于时下热门的分布式计算了。

4.1 分布式计算与 Apache Spark 简介

在计算机科学中，分布式计算（Distributed Computing）是一种计算方法，和传统的集中式计算是相对的。随着计算技术的快速发展，以及计算机所需处理的数据量的激增，有些应用需要消耗非常巨大的计算能力才能完成。如果采用集中式计算，这些应用需要耗费相当长的时间来完成。而分布式计算将这些应用分解成许多小的部分，利用网络分配给多台计算机进行处理。这样可以节约整体计算时间，大大提高计算效率。

Apache Spark 是一个开源的通用分布式计算框架，最初由加州大学伯克利分校的 AMP 实验室开发，可以用来构建大型的、低延迟的分布式计算应用程序。Apache Spark 为 Scala、Java 和 Python 程序设计人员提供了一套高层的面向数据结构的应用程序编程接口（API），它被称为弹性分布式数据集（Resilient Distributed Dataset, RDD）[18]，是一组分布于集群上的可容错只读数据集。我们将利用 Apache Spark 构建分布式计算程序 Hyperspark，它将用于生成所有从超立方体到环的编号方式，以及计算它们的线长。

4.2 基本算法

我们很容易就能注意到，一个从超立方体到环的编号方式在实质上就是一个 0 到 2^d-1 这 2^d 个数的排列。根据这一事实，我们可以得到我们的第一个算法 GeneratePermutationsBySwap，见算法 4.1。

算法 4.1 GeneratePermutationsBySwap

输入： $\{A_i\}_{i=1}^{2^d}$ ▷ 一个给定的初始排列
输入： $start, end$ ▷ 需要生成全排列的开始、结束位置

```
1: if start > end then
2:     Let wirelength = CalculateWirelength(A);        ▷ 计算线长，见算法 4.2
3:     if wirelength ≤ minimum then
4:         Let minimum = wirelength;
5:         Add A to S;                                  ▷ S 是结果集
6:     end if
7: else
8:     for i in start, ..., end do
9:         Swap A_i with A_start;
10:        GeneratePermutationsBySwap(A, start + 1, end);   ▷ 递归
11:        Swap A_i with A_start;
12:    end for
13: end if
```

算法 4.1 首先需要三个输入：$\{A_i\}_{i=1}^{2^d}$ 表示一个给定的初始排列，如 $0, 1, \ldots, 2^d-1$，算法将从该排列开始依次生成每一个排列；$start$ 和 end 则表示本次调用所需要交换的数的位置范围。实质上该算法是一种回溯算法，只是我们还没有对它运用任何剪枝策略（我们会在第 4.3.2 小节讨论如何优化）。对于生成全排列，我们可以这样调用：

$$\text{GeneratePermutationsBySwap}(A, 1, 2^d);$$

算法 4.1 中我们调用了算法 CalculateWirelength 计算一个排列的线长。下面我们给出它的伪代码，见算法 4.2。

算法 4.2 CalculateWirelength

输入： $\{A_i\}_{i=1}^{2^d}$ ▷ 需要计算线长的排列
输出： $wirelength$ ▷ 线长

```
Let wirelength = 0;
for i in 1, ..., 2^d do
    for j in i + 1, ..., 2^d do
        if HasEdge(A_i, A_j) then                    ▷ 见算法 4.3
            if j − i > 2^{d−1} then
                Let wirelength = wirelength + (j − i − 2^{d−1});
            else
                Let wirelength = wirelength + (j − i);
            end if
        end if
    end for
end for
return wirelength;
```

该算法根据排列中的每一对顶点的值判断这两个顶点是否有边相连，若有则更新线长。其中我们用到了算法 HasEdge，其核心则是判断两个数的按位异或值是否为 2 的幂，见算法 4.3（该技巧来源于 Linux 内核源代码）。

算法 4.3 HasEdge

输入： v_1, v_2 ▷ Q_d 中两个顶点的十进制表示形式
输出： **true** or **false** ▷ 是否有边相连

```
Let xorSum = v_1 bitxor v_2;
if (xorSum ≠ 0) and (xorSum bitand (xorSum − 1) = 0) then
    return true;
else
    return false;
end if
```

因此我们得到一个完整的算法 CalculateMinimumWirelength，见算法 4.4。注意这里我们用了一个已知的事实，即格雷码编号方式 $\mathcal{G}$ 的线长为

$$wl(Q_d, C_n, \mathcal{G}) = 3 \cdot 2^{2d-3} - 2^{d-1} \tag{4-1}$$

并且我们假设它是最小的。

算法 4.4 CalculateMinimumWirelength

输入： d ▷ 超立方体的阶
输出： $minimum$ ▷ 线长的最小值
输出： S ▷ 结果集

1: Let $\{A_i\}_{i=1}^{2^d} = 0, 1, \ldots, 2^d - 1$;
2: Let $minimum = 3 \cdot 2^{2d-3} - 2^{d-1}$;
3: Let $S = \emptyset$;
4: GeneratePermutationsBySwap$(A, 1, 2^d)$; ▷ 见算法 4.1
5: **return** $minimum, S$;

4.3 改进和优化

然而上述算法仅能运行于单机环境中，若要令其适用于分布式集群环境，我们则还需要解决以下问题：

(1) 如何将大任务合理有效地划分为小任务并配给分布式集群的计算节点？

(2) 如何将小任务的计算结果合理有效地收集起来？

除此之外，我们似乎也应该考虑计算过程的优化，例如在上一节提到的剪枝策略等。

4.3.1 并行化和 MapReduce

对于问题 1 和 2，Apache Spark 已为我们提供了一套编程模型，它被称为 MapReduce。MapReduce 最初来自于 Jeff Dean 的一篇论文 [12]，它由概念 Map（映射）和 Reduce（归约）构成。一个典型的 MapReduce 程序包含一个 Map 过程，即对数据集里的每一个元素做单独处理；和一个 Reduce 过程，即对数据集里的元素做迭代计算。这两个过程被设计为是可以高度并行的，这对高性能要求的应用以及并行/分布式计算领域的需求来讲非常有用。关于 MapReduce，我们在这里只做上述简要的介绍。如要获取 MapReduce 的更详细信息，请参考 [12]。下面我们就来描述如何利用 Apache Spark 的 RDD 提供的 Map 和 Reduce 接口来解决问题 1 和问题 2。

首先我们可以把 Map 过程理解为主节点（Master Node）将任务的输入数

据进行划分，将每一个部分映射到各个从节点（Slave Node）并让从节点对其进行单独处理。对于我们的程序而言，输入数据为超立方体的阶 d，或者也可以认为是 $2^d!$ 个不同的排列。而直接对 $2^d!$ 个排列进行划分不但会大量消耗主节点的计算资源，而且会造成巨大的主从节点之间的网络传输。为了提高主节点的计算效率，减小网络传输，我们设计了一个用于 Map 过程的算法：它用一个整数——我们称之为*索引*（Index）——来表示一个或多个排列，主节点只需向不同的从节点传输一个不同的索引便可完成主从任务分配，接下来从节点按照算法便可将分配到的索引还原为排列；生成索引序列也非常简单，按照字典序（Lexicographic Order）生成即可。于是对于主节点，我们有了算法 GenerateIndices，见算法 4.5；对于从节点，我们有了算法 RestoreIndexToPermutation，见算法 4.6。为了简洁起见，所展示的算法将本小节的结论与下一小节（第 4.3.2 小节）的结论进行了合并，读者可先阅读下一小节。

算法 4.5 GenerateIndices

输入： $length$ ▷ 需要生成的初始排列的长度

1: Let $nodes = \lfloor \frac{length-1}{2} \rfloor !$;
2: Let $\{I_i\}_{i=1}^{nodes} = 0, 1, \ldots, nodes - 1$;
3: Parallelize(I); ▷ Parallelize 是由 Apache Spark 提供的并行化原语

然后我们考虑 Reduce 过程。Reduce 过程可被理解为主节点向各个从节点收集任务的输出，并做一定的处理。这一阶段 Apache Spark 已为我们做好了绝大多数的工作，我们只需要将各个从节点的计算结果集进行合并即可。

4.3.2 剪枝和去重

下面我们再来探索计算过程中可以采取的优化策略。前文中我们已经指出，算法 4.1 本质上是一个回溯算法，因此我们可以对其进行剪枝，即在每一次递归调用之前加入判断条件，对于那些不符合条件的情况，我们直接跳过该次递归。剪枝策略在某些场景下有很好的优化效果，然而在我们当前这个场景下，我们该如何合理地设计这个判断条件呢？

在给出我们的剪枝策略之前，我们先来讨论一下目标图 C_n 的一些性质。首先我们发现 C_n 上的双射函数——也称为*置换*（Substitution）——是有对称性的。

算法 4.6 RestoreIndexToPermutation

输入： $index$ ▷ 代表一个或多个排列的索引
输入： $length$ ▷ 需要生成的初始排列的长度
输出： $\{L_i\}_{i=1}^{length}$ ▷ 初始排列

Let $\{L_i\}_{i=1}^{3} = 0, 1, 2$;
Let $\{p_i\}_{i=1}^{length} = 0, 0, \ldots, 0$;
for i **in** $length - 1, length - 2, \ldots, 3$ **do**
    Let $p_{i+1} = 2 + index \pmod{i}$; ▷ 0 的位置总是固定在 1，因此加 2
    Let $index = \lfloor \frac{index}{i} \rfloor$;
end for
for i **in** $3, 4, \ldots, length - 1$ **do**
    Insert i into L at position p_{i+1};
end for
return L;

令 $\mathcal{S}$ 表示 C_n 上所有的双射函数构成的集合。很显然，$\mathcal{S}$ 是一个 n 元对称群。

令 $\sigma_r \in \mathcal{S}$ 表示 $\mathcal{S}$ 中的一个置换，满足

$$\sigma_r(i) = (i + 1) \bmod n, i \in V_{C_n} \tag{4-2}$$

则由 σ_r 我们可以生成一个 $\mathcal{S}$ 的子集

$$\mathcal{S}_r = \left\{\sigma_r, \sigma_r^2, \sigma_r^3, \ldots\right\} \tag{4-3}$$

我们称之为*旋转对称*（Rotational Symmetry）。σ_r 的示意图见图 4-1。

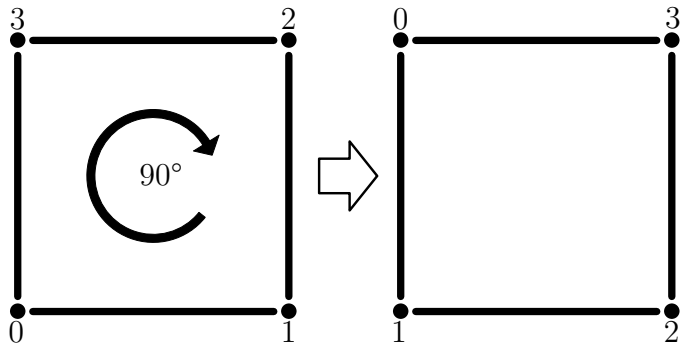


图 4-1　σ_r 下的旋转对称

再令 $\sigma_f \in \mathcal{S}$ 表示 $\mathcal{S}$ 中的一个置换，满足

$$\sigma_f(i) = 2^d - 1 - i, i \in V_{C_n} \tag{4-4}$$

则由 σ_r 和 σ_f 我们可以生成另一个 $\mathcal{S}$ 的子集

$$\mathcal{S}_f = \left\{\sigma_r \cdot \sigma_f, \sigma_r^2 \cdot \sigma_f, \sigma_r^3 \cdot \sigma_f, \dots\right\} \tag{4-5}$$

我们称之为**翻转对称**（Flipping Symmetry）。σ_f 的示意图见图 4-2。注意 $|\mathcal{S}_r| = |\mathcal{S}_f| = n = 2^d$，因为 $\sigma_r^{n+1} = \sigma_r$。

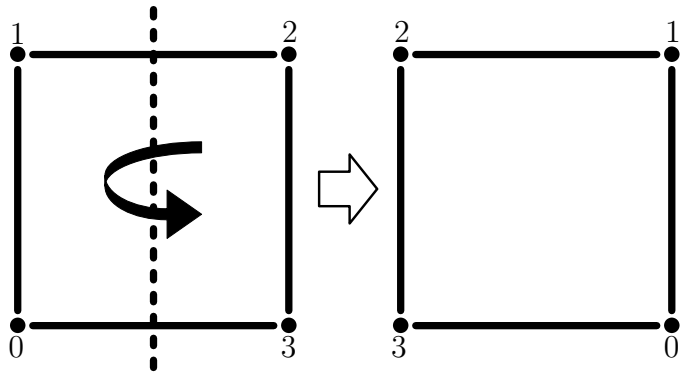


图 4-2　σ_f 下的翻转对称

我们断言，对于任意的编号方式 $\eta\colon V_{Q_d} \to V_{C_n}$ 和任意的置换 $\sigma \in \mathcal{S}_r \cup \mathcal{S}_f$，

$$wl(Q_d, C_n, \eta) = wl(Q_d, C_n, \eta \cdot \sigma) \tag{4-6}$$

有了上述这个结论，我们提出了一个新的生成全排列的算法 GeneratePermutationsByInsertion，见算法 4.7。该算法通过往已经存在 i 个编号的排列中不重复地插入第 $i+1$ 个编号的方式来生成不带有旋转对称和翻转对称这两种“重复情况”的全排列。应用该算法，我们将程序复杂度从原先的 $O\left(2^d!\right)$ 降低至 $O\left(\frac{(2^d-1)!}{2}\right)$。

回到前文讨论的剪枝策略，我们注意到该算法同样也是一个回溯算法。然而与先前不同的是，我们可以很方便地对其进行剪枝：在下一次递归之前计算一次已在排列中的编号的已有线长，若其已不小于已知的最小值，我们就跳过这一次递归，剪去该分支。应用该剪枝策略，我们程序的效率得到了极大的提升（计算 4 阶的等效运行时间大约由原先的 3360 小时左右缩短至了 1 小时左右）。

另外，我们还可以更进一步地改进算法 CalculateWirelength，即算法 4.2。我们可以在计算线长的过程中每更新一次线长就将其与已知的最小值进行比较，

算法 4.7 GeneratePermutationsByInsertion

输入： $element$ ▷ 待插入的元素
输入： $\{L_i\}_{i=1}^{element}$ ▷ 待插入元素的排列

```
1: if element ≥ 2^d then
2:     Add L to S;                                          ▷ S 是结果集
3: else
4:     for i in 2, 3, ..., element + 1 do
5:         Insert element into L at position i;
6:         if CalculateWirelength2(L) ≠ −1 then   ▷ 计算线长来剪枝，见算法 4.8
7:             GeneratePermutationsByInsertion(element + 1, L);   ▷ 递归
8:         end if
9:         Remove element from L at position i;
10:    end for
11: end if
```

若不小于，则不继续进行计算，而是直接返回一个中断标识。我们将改进后的算法称为 CalculateWirelength2，见算法 4.8。

算法 4.8 CalculateWirelength2

输入： $\{L_i\}_{i=1}^{length}$ ▷ 需要计算线长的排列
输出： $wirelength$ ▷ 线长

```
1: Let wirelength = 0;
2: for i in 1, 2, ..., length do
3:     for j in i + 1, i + 2, ..., length do
4:         if HasEdge(L_i, L_j) then                        ▷ 见算法 4.3
5:             Let arclength = j − i > 2^(d−1) ? j − i − 2^(d−1) : j − i;
6:             Let wirelength = wirelength + arclength;
7:             if wirelength > minimum then
8:                 return −1;                               ▷ 返回 −1 表示中断
9:             end if
10:        end if
11:    end for
12: end for
13: return wirelength;
```

最终，我们得到了完整的并行化算法 CalculateMinimumWirelength2，见算法 4.9。

算法 4.9 CalculateMinimumWirelength2

输入： d ▷ 超立方体的阶
输入： $length$ ▷ 初始排列的长度
输出： $minimum$ ▷ 线长的最小值
输出： S ▷ 结果集

1: Let $minimum = 3 \cdot 2^{2d-3} - 2^{d-1}$;
2: Let $S = \emptyset$;
3: GenerateIndices($length$); ▷ 生成和并行化索引序列，见算法 4.5
4: **for all** $index$ **in parallel do**
5: Let $\{L_i\}_{i=1}^{length} =$ RestoreIndexToPermutation($index, length$); ▷ 见算法 4.6
6: GeneratePermutationsByInsertion($length, L$); ▷ 见算法 4.7
7: **end for**
8: **return** $minimum, S$;

上述算法均为 Hyperspark 实现时所用算法。Hyperspark 的核心源代码参见附录 A。

第 5 章 结果分析

利用在第 4 章构建好的分布式计算程序 Hyperspark，我们对 3 阶和 4 阶超立方体到环的嵌入进行了计算。

5.1 示例说明

我们在前文中提到过，一个从超立方体到环的编号方式可以被视为一个 0 到 2^d-1 这 2^d 个数的排列。因此下文中给出的所有结果都将以排列的形式展现出来。在这里我们以 3 阶格雷码编号方式 $\mathcal{G}\colon V_{Q_3}\to V_{C_8}$ 为例，将 $\mathcal{G}$ 表示为

$$\mathcal{G}=[0,1,3,2,6,7,5,4] \tag{5-1}$$

其中每一个数是 Q_3 上每一个顶点的十进制表示形式。该排列表示 Q_3 中的顶点 000（十进制表示为 0）映射到 C_8 中的顶点 0（第一个位置）、Q_3 中的顶点 001（十进制表示为 1）映射到 C_8 中的顶点 1（第二个位置）、……、Q_3 中的顶点 100（十进制表示为 4）映射到 C_8 中的顶点 7（第八个位置）。

5.2 3 阶情况

首先是 3 阶的计算结果。3 阶的情况下一共只有 $40,320$ 种编号方式，在如今的计算机的运算能力下仅依靠未做任何优化的单机计算程序（如算法 4.4）就可以在瞬间得到结果。因此，我们很自然地选择了 3 阶情况的计算作为验证 Hyperspark 正确性的测试；而对于程序的运行时间等其他指标，我们在这次计算中将选择性地忽略。

我们先运行算法 4.9 的实现，即 Hyperspark。调用

$$\text{CalculateMinimumWirelength2}(3,3);$$

其中输入数据为 $d=3$ 和 $length=3$，表示计算 3 阶的情况，并设定初始排列的长度为 3。初始排列的长度决定了我们要将任务划分成多少份，即要把小任务交给多少个从节点进行计算，其对应关系为

$$length\to\frac{(length-1)!}{2} \tag{5-2}$$

因为 $0,1,2$ 的不带重复的环形排列只有 1 个，而要往一个 $0,1,\ldots,x-1$ 的环形排列中插入 x 则有 x 个不带重复的位置可供插入。

通过上述调用，我们最终得到 3 阶情况下的最短线长为 20，以及 48 个 3 阶排列：

$$
\begin{array}{llll}
[0,6,4,7,5,3,1,2], & [0,6,4,5,7,3,1,2], & [0,4,6,7,5,3,1,2], & [0,4,6,5,7,3,1,2],\\
[0,3,1,7,5,6,4,2], & [0,3,1,5,7,6,4,2], & [0,3,1,7,5,4,6,2], & [0,3,1,5,7,4,6,2],\\
[0,6,4,7,5,1,3,2], & [0,6,4,5,7,1,3,2], & [0,4,6,7,5,1,3,2], & [0,4,6,5,7,1,3,2],\\
[0,4,5,1,7,3,6,2], & [0,4,5,1,3,7,6,2], & [0,4,5,1,7,3,2,6], & [0,4,5,1,3,7,2,6],\\
[0,4,1,5,7,3,6,2], & [0,4,1,5,3,7,6,2], & [0,4,1,5,7,3,2,6], & [0,4,1,5,3,7,2,6],\\
[0,1,5,4,7,6,3,2], & [0,1,5,4,6,7,3,2], & [0,1,4,5,7,6,3,2], & [0,1,4,5,6,7,3,2],\\
[0,1,3,7,5,6,4,2], & [0,1,3,5,7,6,4,2], & [0,1,3,7,5,4,6,2], & [0,1,3,5,7,4,6,2],\\
[0,5,1,7,3,6,2,4], & [0,5,1,3,7,6,2,4], & [0,5,1,7,3,2,6,4], & [0,5,1,3,7,2,6,4],\\
[0,1,5,7,3,6,2,4], & [0,1,5,3,7,6,2,4], & [0,1,5,7,3,2,6,4], & [0,1,5,3,7,2,6,4],\\
[0,1,3,2,7,6,5,4], & [0,1,3,2,6,7,5,4], & [0,1,3,2,7,6,4,5], & [0,1,3,2,6,7,4,5],\\
[0,1,5,4,7,6,2,3], & [0,1,5,4,6,7,2,3], & [0,1,4,5,7,6,2,3], & [0,1,4,5,6,7,2,3],\\
[0,1,2,3,7,6,5,4], & [0,1,2,3,6,7,5,4], & [0,1,2,3,7,6,4,5], & [0,1,2,3,6,7,4,5]
\end{array}
$$

注意我们已经在生成全排列的算法中排除了旋转对称和翻转对称这两种“重复情况”，因此实际情况中的最优排列数肯定会大于 48。并且我们可以对其进行估算：长为 n 的全排列有 $2n$ 种对称情况，正如我们在第 4.3.2 小节所见；反过来，知道了不带重复的最优排列数，我们可以估算总的最优排列数为不带重复的最优排列数乘以 $2n$。也就是说我们估算 3 阶情况下的最优排列数应该是 $48 \times 2 \times 8 = 768$。

我们接下来运行算法 4.4 的实现。调用

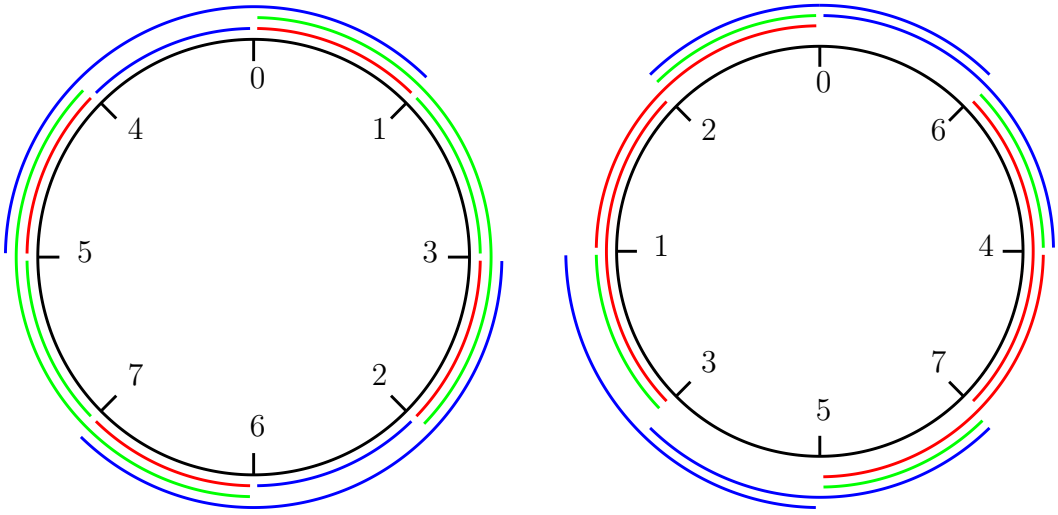
```
CalculateMinimumWirelength(3);
```

其中输入数据为 $d = 3$，表示计算 3 阶的情况。

通过上述调用，我们得到 3 阶情况下的最短线长为 20，以及 768 个 3 阶排列（由于数据量过大，我们不在这里列出），从而印证了我们的估算。同时这也说明了我们 Hyperspark 的正确性，以及在 3 阶情况下 Guu 的结论的正确性。

下面我们对这 48 个排列进行简单的数据分析。为此我们开发了一个辅助性的 Matlab 画图程序，用来把每一个排列和它的线长分布情况以图形的方式

清晰直观地展现出来。该程序的核心源代码参见附录 B。由于篇幅限制，我们从 48 个排列中随机选择了两个排列的图形放在本文中，一个是格雷码编号方式，见图 5-1a；另一个则是我们结果集中的第一个编号方式，见图 5-1b。



(a) 格雷码编号方式　　(b) 另一种最优编号方式

图 5-1　Q_3 到 C_8 的两种编号方式

上面两个图中的黑色的圆表示我们的目标图 C_8。C_8 的顶点从圆上最顶端的位置起沿顺时针方向依次分布在圆上，我们用一条指向圆心的黑色线段来标识。对于 Q_3，我们用数字 $0, 1, \ldots, 7$——即十进制表示形式——来表示它的顶点，用黑色圆外的每一段彩色圆弧来表示它的每一条边。

注意我们将 Q_3 的每一条边用不同的颜色进行了区分，而区分的根据则是一条边对应的两个顶点的哪一位不同。回想一下，在超立方体 Q_d 中，两个顶点 $v, w \in V_{Q_d}$ 之间有边相连当且仅当 v 和 w 的坐标（二进制表示形式）正好只有一位不相同。在这里，我们用红色的圆弧表示第 1 位（最低位）坐标不相同的边，用绿色的圆弧表示第 2 位（次低位）坐标不相同的边，接下来是蓝色和粉色（粉色在 3 阶情况中不存在，但存在于接下来的 4 阶情况中）。

通过画图程序可视化所有 48 个排列后，我们发现 3 阶情况下的最优排列有一个共同的特点：每一个最优排列都存在 4 条 C_8 中的边，它们的边拥塞度为 2，并且它们的位置也是相对固定的，即若找到一条这样的边，我们可以马上找到其余的三条；而其他剩下的 4 条边每一条都有边拥塞度大于 2。不仅如

此，我们还发现包含边拥塞度为 2 的 Q_3 中的边只对应同一种颜色，而包含边拥塞度大于 2 的 Q_3 中的边则至少对应两种颜色。

用更严格的形式来表示，对于 $\forall e \in E_{Q_d}$ 且 $\partial(e) = \{v, w\}$，我们令

$$t(e) = i, \text{ 其中 } v_i \neq w_i, 1 \leq i \leq d \tag{5-3}$$

表示 Q_d 中的边 e 的类型。那么对于每一个最优编号方式 η: $V_{Q_3} \to V_{C_8}$，

(1) 存在 4 条边 $e_1, e_2, e_3, e_4 \in E_{C_8}$ 满足对于 $\forall k \in \{1, 2, 3, 4\}$，

$$|EC_\eta(e_k)| = 2 \tag{5-4}$$

(2) 对于 $\forall k \in \{1, 2, 3, 4\}$ 和 $\forall e'_1, e'_2 \in EC_\eta(e_k)$，如果同时有 $e_k \in P(e'_1)$ 和 $e_k \in P(e'_2)$，那么

$$t(e'_1) = t(e'_2) \tag{5-5}$$

(3) 若令 $e_k = \{i_k, j_k\}$，$1 \leq k \leq 4$，那么有

$$\begin{cases} i_1 \equiv (i_2 - 2) \bmod 8 \equiv (i_3 - 4) \bmod 8 \equiv (i_4 - 6) \bmod 8 \\ j_1 \equiv (j_2 - 2) \bmod 8 \equiv (j_3 - 4) \bmod 8 \equiv (j_4 - 6) \bmod 8 \end{cases} \tag{5-6}$$

我们把式 (5-4) 称为边拥塞公式，把式 (5-5) 称为同色公式，把式 (5-6) 称为位置公式。

注意这三个公式只适用于 Q_3 到 C_8 的最优编号方式，对于这一特性是否在 4 阶情况下也保持成立，我们将在下一节中进行探索。

5.3 4 阶情况

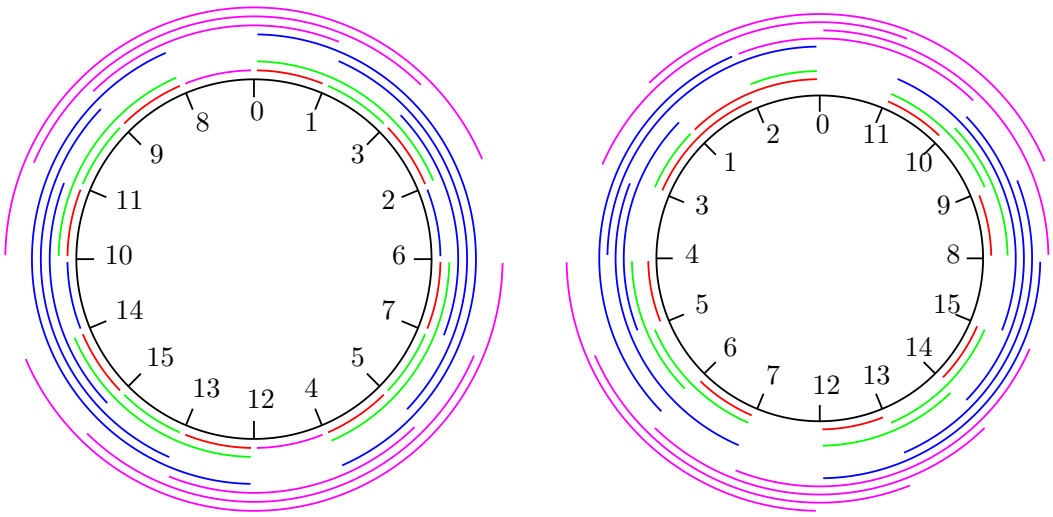
下面是 4 阶的计算结果。4 阶的情况下一共只有 16! 种编号方式，因此算法 4.4 已不再适用了。我们直接运行 Hyperspark，调用

CalculateMinimumWirelength2(4, 4);

其中输入数据为 $d = 4$ 和 $length = 4$。$length = 4$ 的条件下主节点将生成 $\frac{3!}{2} = 3$ 个索引并分配给 3 个从节点，每个从节点将各自的索引还原为长度为 4 的不同的初始排列，再用带剪枝的插入法生成不带重复的全排列。

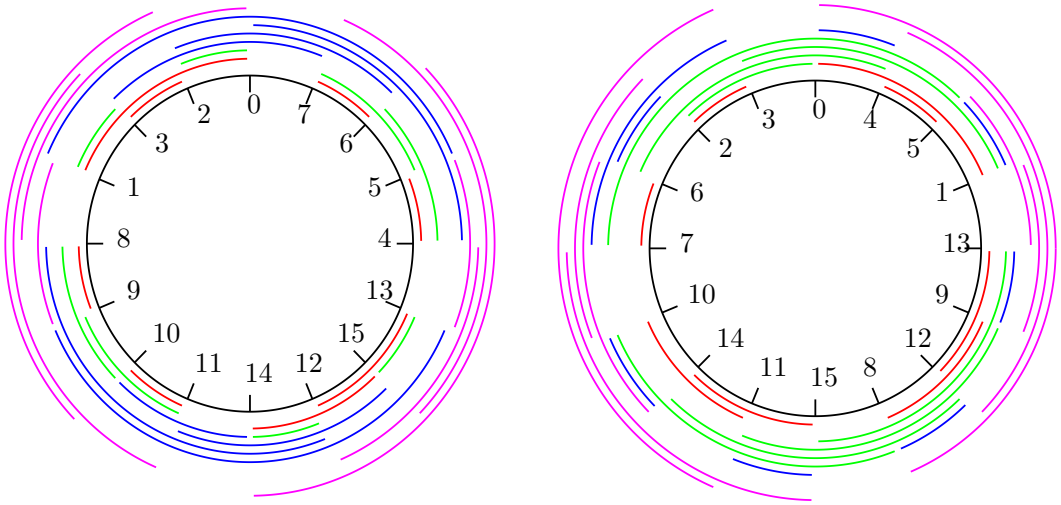
通过上述调用，我们最终得到 4 阶情况下的最短线长为 88（说明在 4 阶情况下 Guu 的结论也是正确的），以及来自从节点的 3 个结果集，我们分别称为

结果集 1、2 和 3。其中结果集 1 和 3 各自包含了 81,920 个最优排列，结果集 2 则包含了 229,376 个最优排列（因此我们可以估算总的最优排列数为 $(81920 \times 2 + 229376) \times 2 \times 16 = 12,582,912$，在这里我们不做验证）。我们从每个结果集中分别挑选 1 个最优编号方式的图形放在本文中，外加格雷码编号方式（它存在于结果集 2 中），见图 5-2 和图 5-3。



(a) 格雷码编号方式　　(b) 结果集 1 中的一个编号方式

图 5-2　Q_4 到 C_{16} 的两种编号方式



(a) 结果集 2 中的一个编号方式　　(b) 结果集 3 中的一个编号方式

图 5-3　Q_4 到 C_{16} 的另外两种编号方式

我们惊奇地发现，我们在上一节中提出的特性不但适用于 3 阶情况，也同样适用于 4 阶情况：每一个最优排列都存在 4 条 C_{16} 中的边，它们的边拥塞度为 4，位置相对固定；其他剩下的 12 条边每一条都有边拥塞度大于 4。包含边拥塞度为 4 的 Q_4 中的边只对应同一种颜色，包含边拥塞度大于 4 的 Q_4 中的边至少对应两种颜色。

下面我们给出一个更一般的结论。

猜想 5.1. *对于 $\forall d \geq 2$ 和每一个最优编号方式 $\eta\colon V_{Q_d} \to V_{C_n}$，*

(1) 存在 4 条边 $e_1, e_2, e_3, e_4 \in E_{C_n}$ 满足对于 $\forall k \in \{1,2,3,4\}$，

$$|EC_\eta(e_k)| = 2^{d-2} \tag{5-7}$$

(2) 对于 $\forall k \in \{1,2,3,4\}$ 和 $\forall e_1', e_2' \in EC_\eta(e_k)$，如果同时有 $e_k \in P(e_1')$ 和 $e_k \in P(e_2')$，那么

$$t(e_1') = t(e_2') \tag{5-8}$$

(3) 若令 $e_k = \{i_k, j_k\}$，$1 \leq k \leq 4$，那么有

$$\begin{cases} i_1 \equiv (i_2 - 2^{d-2}) \bmod n \equiv (i_3 - 2^{d-1}) \bmod n \equiv (i_4 - 3 \cdot 2^{d-2}) \bmod n \\ j_1 \equiv (j_2 - 2^{d-2}) \bmod n \equiv (j_3 - 2^{d-1}) \bmod n \equiv (j_4 - 3 \cdot 2^{d-2}) \bmod n \end{cases} \tag{5-9}$$

对于该结论是否在 $d \geq 5$ 时成立，我们还无法对其进行验证。然而我们相信这个结论是正确的，因为 5 阶格雷码编号方式 $\mathcal{G}\colon V_{Q_5} \to V_{C_{32}}$ 依然满足这三个式子。

第 6 章 结论

纵观全文，我们在本文中研究了从超立方体到环的嵌入的一些性质。首先我们介绍了图嵌入领域——更具体地说，则是从超立方体到链、环等结构的嵌入问题——的历史研究和进展，包括由 L. H. Harper 和 A. J. Bernstein 解决的边等周问题 [7,11]、C. Guu 提出的派生网络 [3]，以及 P. Manuel 提出的边拥塞理论 [4] 等，并对边等周问题的证明提出了改进。接下来我们引入了分布式计算，给出了分布式计算程序 Hyperspark 所用到的算法和一些理论基础，利用工程的方法来验证和探索从超立方体到环嵌入的一些重要性质，并根据得到的结果集提出了最优嵌入的一些新的特性。

根据我们的结果，Guu 的结论——即从超立方体到环的嵌入在格雷码编号方式下有最小的线长——在 3 阶和 4 阶情况下都是正确的。由于有限的计算资源，以及该问题自身的限制，我们无法对 5 阶及以上的情况进行验证。而在新特性方面，利用 Matlab 画图程序对 Hyperspark 的结果集进行数据可视化后，我们找到了一些最优编号方式所共有的性质，并对其进行了归纳和总结，见猜想 5.1。

6.1 未来工作

针对从超立方体到环嵌入这一问题，以及我们的研究结果，未来还有许多进一步的工作可以继续开展：

(1) 首先，我们可以继续进行图嵌入理论上的研究。对于 5 阶及以上的情况，由于巨大的数据量，现今任何工程上的方法都不可能一劳永逸地解决该问题。我们相信理论上存在一种方法能被给出用来证明该问题，亦或是修补 Guu 的证明。

(2) 对于我们提出的猜想 5.1，我们并没有给出它的证明，因此在下一步的工作中我们可以先对该猜想进行证明（或证伪），并探究最后的结论对于该问题的解决是否有意义。

(3) 在运行 4 阶情况时，我们的程序产生了大量数据（结果集中的排列），然而我们没有对其进行完全分析。因此在未来我们可以利用适当的统计分析方法和更加先进的数据分析工具，对我们的数据加以详细研究和概括总结。

致谢

首先，我要特别感谢我的导师——刘庆晖老师。从最开始的毕设选题，直至最后的论文定稿，我正是在他的引领下一步一步走完这本科四年从未体验过的学术之旅。他在数学和计算机领域广博的学识对我的论文的完成有着非常大的帮助，而他严谨的治学精神和一丝不苟的工作作风也让我受益匪浅。

其次，我要感谢我的室友，以及所有帮助过我的老师、同学，感谢他们的包容与体谅，关心与支持。

最后，我要感谢我的家人，感谢他们在我的本科四年中源源不断地给予我爱和精神上的支持。我所取得的成绩离不开他们在我背后默默的奉献。

参考文献

[1] K. H. Rosen. *Discrete Mathematics and Its Applications 7th edition* [M]. McGraw-Hill Education, 2012. 641-744.

[2] L. H. Harper. *Global Methods for Combinatorial Isoperimetric Problems* [M]. Cambridge University Press, 2004. 1-19.

[3] C. Guu. *The Circular Wirelength Problem for Hypercubes* [D]. California: University of California, Riverside, 1997.

[4] P. Manuel, I. Rajasingh, B. Rajan, H. Mercy. *Exact wirelength of hypercubes on a grid* [J]. Discrete Applied Mathematics, 2009, 157(7): 1486-1495.

[5] P. Manuel, M. Arockiaraj, I. Rajasingh, B. Rajan. *Embedding hypercubes into cylinders, snakes and caterpillars for minimizing wirelength* [J]. Discrete Applied Mathematics, 2011, 159(17): 2109-2116.

[6] J. Opatrny, D. Sotteau. *Embeddings of complete binary trees into grids and extended grids with total vertex-congestion 1* [J]. Discrete Applied Mathematics, 2000, 98(3): 237-254.

[7] L. H. Harper. *Optimal assignments of numbers to vertices* [J]. Journal of the Society for Industrial and Applied Mathematics, 1964, 12(1): 131-135.

[8] J. Erbele, J. Chavez, R. Trapp. *The Cyclic Cutwidth of Q_n* [J/OL]. California State University, San Bernardino, 2003.

[9] S. L. Bezrukov, J. D. Chavez, L. H. Harper, M. Röttger, U.-P. Schroeder. *The Congestion of n-Cube Layout on a Rectangular Grid* [J]. Discrete Mathematics, 2000, 213(1-3): 13-19.

[10] W. Ji, Q. Liu, G. Wang, Z. Shen. *Embedding of Hypercube into Cylinder* [J/OL]. arXiv: 1511.07932, 2015.

[11] A. J. Bernstein. *Maximally connected arrays on the n-cube* [J]. SIAM Journal on Applied Mathematics, 1967, 15(6): 1485-1489.

[12] J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters* [J]. Communications of the ACM, 2008, 51(1): 107-113.

[13] L. H. Harper. *Optimal Numberings and Isoperimetric Problems on Graphs* [J]. Journal of Combinatorial Theory, 1966, 1(3): 385-393.

[14] K. Steiglitz, A. J. Bernstein. *Optimal Binary Coding of Ordered Numbers* [J]. Journal of the Society for Industrial and Applied Mathematics, 1965, 13(2): 441-443.

[15] J. D. Chavez, R. Trapp. *The cyclic cutwidth of trees* [J]. Discrete Applied Mathematics, 1998, 87(1-3), 25-32.

[16] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness* [M]. W. H. Freeman and Company, 1979. 1-338.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. *Spark: Cluster Computing with Working Sets* [A]. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing [C]. Boston, MA: USENIX Association, 2010. 10-10.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing* [A]. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation [C]. San Jose, CA: USENIX Association, 2012. 15-28.

附录 A Hyperspark 核心源代码

A.1 App.java

```
package me.chaosdefinition.hyperspark;

import java.util.List;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

import me.chaosdefinition.hyperspark.common.HypersparkException;
import me.chaosdefinition.hyperspark.core.WirelengthCalculator;
import me.chaosdefinition.hyperspark.optparse.OptionParser;
import me.chaosdefinition.hyperspark.util.MathUtils;

/**
 * Application starting point.
 *
 * @author Chaos Shen
 */
public class App {
  public static void main(String[] args) {
    try {
      /* Parse command line options. */
      OptionParser parser = new OptionParser();
      parser.parse(args);

      /* Get options. */
      int dimension = parser.getDimension();
      int length = parser.getLength() < 3 ? 3 : parser.getLength();
      String outdir = parser.getOutdir();
      boolean verbose = parser.isVerbose();

      /* Initialize Spark environment. */
      SparkConf conf = new SparkConf().setAppName("hyperspark");
      JavaSparkContext ctx = new JavaSparkContext(conf);

      /*
       * Generate a list of lexicographic ordered integers, each
```

```
 * representing an index in all permutations of specified length
 * without rotation and flip symmetries, and then parallelize them
 * to JavaRDD.
 */
JavaRDD<Integer> indices = ctx.parallelize(
    MathUtils.lexicode(MathUtils.factorial(length - 1) / 2)
);

if (verbose) {
  /*
   * In verbose mode, first convert each index to corresponding
   * initial permutation, and then construct a
   * WirelengthCalculator with specified dimension and the initial
   * permutation, finally call findMinimumMappings() which returns
   * the result by this permutation.
   *
   * We don't need to reduce all the results since they are
   * exactly what we want. Save them to the output directory.
   */
  indices.flatMap(index -> {
    List<Integer> initial = MathUtils.indexToPermutation(
      index, length
    );
    WirelengthCalculator calculator = new WirelengthCalculator(
      dimension, initial
    );
    calculator.setVerbose(verbose);
    return calculator.findMinimumMappings();
  }).saveAsTextFile(outdir);
} else {
  /*
   * Otherwise, we just need to know the minimum wirelength, which
   * can be acquired by following the above steps but finally
   * calling calculateMinimumWirelength() instead.
   *
   * Reduce the results by taking the minimum and print the final
   * minimum.
   */
  int minimum = indices.map(index -> {
    List<Integer> initial = MathUtils.indexToPermutation(
      index, length
    );
    WirelengthCalculator calculator = new WirelengthCalculator(
```

```
            dimension, initial
          );
          return calculator.calculateMinimumWirelength();
        }).reduce((v1, v2) -> Math.min(v1, v2));
        System.out.println("Minimum: " + minimum);
      }
      ctx.close();
    } catch (Exception e) {
      if (e instanceof HypersparkException) {
        e.printStackTrace();
      } else {
        new HypersparkException(e).printStackTrace();
      }
      System.exit(1);
    }
  }
}
```

A.2 WirelengthCalculator.java

```
package me.chaosdefinition.hyperspark.core;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import me.chaosdefinition.hyperspark.common.HypersparkException;

/**
 * The main class of hyperspark.
 *
 * @author Chaos Shen
 */
public class WirelengthCalculator {

  private int dimension;
  private int vertices;
  private List<Integer> initial;
  private int minWirelength;
  private List<List<Integer>> minMappings;

  private boolean verbose = false;

  /**
   * Determines if two vertices have an edge between them.
   *
   * @param v1
   *            the integer representation of a vertex
   * @param v2
   *            the same as {@code v1}
   * @return {@code true} if the two vertices have an edge between them,
   *         otherwise {@code false}
   */
  public static boolean hasEdge(int v1, int v2) {
    /*
     * An egde exists if only one bit differs, so we do this check by
     * determing if their xor sum is a power of two.
     */
    return (v1 ^ v2) != 0 && ((v1 ^ v2) & ((v1 ^ v2) - 1)) == 0;
```

```
}

/**
 * Returns the known minimum wirelength of circular mappings of
 * hypercube of specified dimension, which typically is the
 * wirelength of the graycode mapping.
 *
 * @param dimension
 *            the dimension of the hypercube
 * @return the known minimum circular wirelength
 */
public static int knownMinimumWirelength(int dimension) {
  if (dimension < 0) {
    throw new HypersparkException(
      "Illegal value of dimension: " + dimension
    );
  } else if (dimension < 2) {
    return dimension;
  } else {
    return 3 * (1 << ((dimension << 1) - 3)) - (1 << (dimension - 1));
  }
}

/**
 * Constructs a {@link WirelengthCalculator} for hypercube of specified
 * dimension.
 *
 * @param dimension
 *            the dimension of the hypercube
 */
public WirelengthCalculator(int dimension) {
  this(dimension, null);
}

/**
 * Constructs a {@link WirelengthCalculator} for hypercube of specified
 * dimension with a given initial list to start with.
 *
 * @param dimension
 *            the dimension of the hypercube
 * @param initial
 *            the initial list to start with
 */
```

```
public WirelengthCalculator(int dimension, List<Integer> initial) {
  this.dimension = dimension;
  this.vertices = 1 << dimension;
  if (initial != null) {
    /* check duplication of each entry */
    boolean[] presence = new boolean[initial.size()];
    for (int i : initial) {
      if (i >= 0 && i < initial.size()) {
        presence[i] = true;
      }
    }
    for (boolean b : presence) {
      if (!b) {
        throw new HypersparkException(
          "Illegal value in initial list!"
        );
      }
    }
    this.initial = new ArrayList<>(initial);
  } else {
    /* [0, 1, 2] will produce all permutations without symmetry */
    this.initial = new ArrayList<>(Arrays.asList(0, 1, 2));
  }

  this.minWirelength = knownMinimumWirelength(dimension);
}

/**
 * Calculates the minimum circular wirelength on all mappings under the
 * initial list.
 *
 * @return the minimum circular wirelength
 */
public int calculateMinimumWirelength() {
  backtrack(this.initial, this.initial.size());
  return this.minWirelength;
}

/**
 * Finds all the mappings of minimum circular wirelength under the
 * initial list.
 *
 * @return a list of minimum mappings
```

```
 */
public List<List<Integer>> findMinimumMappings() {
  this.minMappings = new ArrayList<>();
  if (!this.verbose) {
    setVerbose(true);
    backtrack(this.initial, this.initial.size());
    setVerbose(false);
  } else {
    backtrack(this.initial, this.initial.size());
  }
  return this.minMappings;
}

/**
 * Calculates the complete circular wirelength of a list.
 *
 * @param list
 *            a list (an initial segment or a complete mapping of
 *            hypercube)
 * @return the complete circular wirelength
 */
public int calculateWirelength(List<Integer> list) {
  return calculateWirelength(list, 0, list.size());
}

/**
 * Calculates the partial circular wirelength of a list.
 *
 * @param list
 *            a list (an initial segment or a complete mapping of
 *            hypercube)
 * @param start
 *            the starting position in the list
 * @param end
 *            the ending position in the list
 * @return the partial circular wirelength, or -1 when partial
 *         wirelength is already larger than the current minimum
 */
public int calculateWirelength(List<Integer> list, int start, int end) {
  int wirelength = 0;

  for (int i = start; i < end; ++i) {
    for (int j = i + 1; j < end; ++j) {
```

```
      if (hasEdge(list.get(i), list.get(j))) {
        if ((j - i) * 2 > this.vertices) {
          wirelength += this.vertices - (j - i);
        } else {
          wirelength += j - i;
        }
        if (wirelength > this.minWirelength) {
          return -1;
        }
      }
    }
  }

  return wirelength;
}

/**
 * Performs backtracking to generate all the mappings under the
 * initial list.
 *
 * @param list
 *            a list (an initial segment or a complete mapping of
 *            hypercube)
 * @param element
 *            next element to add to the list
 */
private void backtrack(List<Integer> list, int element) {
  if (element >= this.vertices) {
    this.minWirelength = calculateWirelength(list);
    if (this.verbose) {
      this.minMappings.add(new ArrayList<>(list));
    }
  } else {
    for (int i = 1; i <= list.size(); ++i) {
      list.add(i, element);
      if (calculateWirelength(list) != -1) {
        backtrack(list, element + 1);
      }
      list.remove(i);
    }
  }
}
```

```
  /* getters and setters */
  public int getDimension() {
    return dimension;
  }

  public int getVertices() {
    return vertices;
  }

  public List<Integer> getInitial() {
    return initial;
  }

  public int getMinWirelength() {
    return minWirelength;
  }

  public boolean isVerbose() {
    return verbose;
  }

  public void setVerbose(boolean verbose) {
    this.verbose = verbose;
  }
}
```

A.3 MathUtils.java

```
package me.chaosdefinition.hyperspark.util;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import me.chaosdefinition.hyperspark.common.HypersparkException;

/**
 * Relevant mathematical utilities.
 *
 * @author Chaos Shen
 */
public class MathUtils {

  /**
   * Generates the factorial of a given integer.
   *
   * @param n
   *            an integer
   * @return its factorial
   */
  public static int factorial(int n) {
    final int[] factorials = {
      1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
      479001600
    };

    if (n >= factorials.length) {
      throw new HypersparkException("Factorial of " + n + "too large!");
    }

    return factorials[n];
  }

  /**
   * Converts an integer to a permutation of given length without
   * rotation and flip symmetries.
   *
```

```
   * @param index
   *            an integer ranging from 0 to ((n - 1)! / 2 - 1)
   * @param length
   *            the length of the target permutation
   * @return the target permutation
   */
  public static List<Integer> indexToPermutation(int index, int length) {
    List<Integer> code = new ArrayList<>(Arrays.asList(0, 1, 2));
    int[] position = new int[length];

    for (int i = length - 1; i >= 3; --i) {
      /* 0 is always fixed, so we need to shift others each by 1 */
      position[i] = index % i + 1;
      index /= i;
    }
    for (int i = 3; i < length; ++i) {
      code.add(position[i], i);
    }

    return code;
  }

  /**
   * Generates the lexicographic code of given length.
   *
   * @param length
   *            the length of the code
   * @return the lexicographic code
   */
  public static List<Integer> lexicode(int length) {
    List<Integer> code = new ArrayList<>(length);
    for (int i = 0; i < length; ++i) {
      code.add(i);
    }
    return code;
  }
}
```

附录 B　Matlab 画图程序核心源代码

B.1　main.m

```
axis equal;
hold on;

% open data file
[file, path] = uigetfile('*');
if file == 0
  quit;
end
fileID = fopen(strcat(path, file));

while 1
  % read a line, each line representing a result
  line = fgetl(fileID);
  if line == -1
    break;
  end
  data = sscanf(line, '%*c%d')';
  total = size(data, 2);
  cla;

  % draw a basic inner circle
  innerTheta = 0 : pi / 10000 : 2 * pi;
  innerX = 1 * sin(innerTheta);
  innerY = 1 * cos(innerTheta);
  plot(innerX, innerY, 'k');

  % draw the marks
  markTheta = 0 : 2 * pi / total : 2 * pi;
  markX = [];
  markY = [];
  for j2 = 1 : total
    markX = [1, 0.9, 0.8] * sin(markTheta(j2));
    markY = [1, 0.9, 0.8] * cos(markTheta(j2));
    plot(markX([1, 2]), markY([1, 2]), 'k');
    text(markX(3), markY(3), num2str(data(j2)));
  end
```

```
  % draw the outside arcs
  counters = zeros(1, total);
  heights = zeros(1, total);
  colors = ['r', 'g', 'b', 'm', 'y', 'c'];
  for i = 0 : log2(total) - 1
    for j = 1 : total / 2
      for k = 1 : total
        if j == 8 && k > 7
          continue;
        end
        j2 = mod(k + j - 1, total) + 1;
        if hasEdge(data(k), data(j2)) && ...
            bitxor(data(k), data(j2)) == 2^i
          [counters, heights] = drawArc(counters, heights, ...
              k, j2, total, ...
              colors(i + 1));
        end
      end
    end
  end

  % listen to keyboard to choose the next action
  while 1
    try
      action = waitforbuttonpress;
      if action
        action = get(gcf, 'CurrentCharacter');
      end
    catch
      action = 'q';
    end
    switch action
    case 'n'
      break;
    case 'q'
      fclose(fileID);
      quit;
    end
  end
end

fclose(fileID);
quit;
```

B.2 drawArc.m

```
%% drawArc: draw an arc
function [counters, heights] = drawArc(counters, heights, i1, i2, ...
    total, color)
  maximum = 0;

  if i1 > i2
    i2 = i2 + total;
  end
  for i = i1 : i2 - 1
    index = mod(i - 1, total) + 1;
    if heights(index) > maximum
      maximum = heights(index);
    end
  end
  maximum = maximum + 1;
  for i = i1 : i2 - 1
    index = mod(i - 1, total) + 1;
    heights(index) = maximum;
    counters(index) = counters(index) + 1;
  end

  radius = 1 + maximum * 0.05;
  delta = pi / 80;
  degree = 2 * pi / total;
  theta = (i1 - 1) * degree + delta : pi / 1000 : (i2 - 1) * degree - delta;
  arcX = radius * sin(theta);
  arcY = radius * cos(theta);
  plot(arcX, arcY, color);
```

B.3 hasEdge.m

```
%% hasEdge: determine if two vertices have an edge between them
function [v] = hasEdge(a, b)
  xorSum = bitxor(a, b);
  v = xorSum ~= 0 & bitand(xorSum, xorSum - 1) == 0;
  return;
```