新竹市第四十四屆中小學科學展覽會

作品說明書

科　　別：　數學科

組　　別：　國中組

作品名稱：「魔」力啟動，滿級進化--魔術方塊 CFOP 和 ZBLL 平均步數之比較

關 鍵 詞：魔術方塊、CFOP、ZBLL

編　　號：

摘要

本研究旨在探討魔術方塊（Rubik's Cube）的專業速解領域中，主流還原系統 CFOP（Fridrich Method）與進階系統 ZBLL（Zborowski-Bruchem Last Layer）在平均還原步數上的差異。

透過數學期望值計算與電腦模擬數據統計分析，比較兩者在完成前兩層（F2L）後的頂層處理效率。

研究結果顯示，標準 CFOP 的頂層還原（OLL + PLL）平均約需 21.5 步（HTM）；而採用 ZBLL 系統平均僅需 14.8 步。然而，若考量 ZBLL 前置的邊塊導向（EO）步驟所增加的額外步數，ZBLL 系統相較於 CFOP 的淨節省步數約為 3-5 步。本研究結論為：ZBLL 確實具有顯著的步數優勢，但須權衡其龐大的公式記憶量與判斷時間成本。

壹、研究動機

魔術方塊(rubic’ s cube)自 1974 年由匈牙利盧比克教授發明以來，速解（Speedcubing）已發展成為一項競技運動。目前世界紀錄不斷刷新，由上世紀的第一個紀錄 22 秒到今年剛在 2/7 號由波蘭 Teodor Zajder 刷新的 2.76 秒，為了能突破極限，選手們除了提升手速（TPS, Turns Per Second），更致力於尋求更短的解法路徑。

雖然魔術方塊是相當常見的題材，但相較於歷年來已經被研究的相當透徹的最短步數解、循環公式及不同異形方塊的比較，我們的研究主要聚焦在三階魔方速解方法之比較--透過系統性的分析，給出對選手最適當的建議。

目前最主流的解法為 CFOP，將還原過程分為四個階段。然而，頂尖選手開始引入 ZBLL 技術，試圖將頂層的兩個步驟合併為一步。我們好奇的是：背誦多達 493 個公式的 ZBLL 系統，究竟能比標準 CFOP 系統節省多少步數？其步數縮減的效益是否符合「記憶成本」的比例？ 這是本研究欲探討的核心問題。

研究目的

一. 用數學分析證明魔術方塊速解的公式總數。

二. 分析傳統主流速解方法-- CFOP 方法中，頂層（Last Layer, LL）雙公式復原處理的平均步數之期望值。

三. 分析 ZBLL 方法中，單公式處理頂層的平均步數期望值。

四. 探討為了執行 ZBLL 所需的邊塊導向前置動作（Edge Orientation, EO）對總步數的影響。

五. 比較兩種系統的理論最小步數與實戰應用差異。

六. ZBLL 和 CFOP 頂層反應次數與時間的比較。

貳、研究設備與器材

一. 標準三階魔術方塊（3x3x3 Rubik's Cube）。

二. 電腦運算設備（用於查閱數據庫與模擬）。

三. Cube Explorer / Cube Solver 軟體（用於驗證最佳步數）。

四. 速解計時器與步數記錄表。

五. 相關文獻與公式列表（CFOP 119 態、ZBLL 493 態）。

六. visual studio C++程式編譯器。

參、研究過程與方法

一. 名詞定義與計量標準

(一)、名詞定義：

1. **邊塊**：位於方塊每一條「邊」中間的方塊。在一個標準的 3x3 魔術方塊中，共有 **12 個** 邊塊。
2. **角塊**：位於魔術方塊 8 個頂角上的零件。在標準的 3x3 魔術方塊中，角塊的數量固定為 **8 個**。
3. **中心塊：位於魔術方塊每個面的正中央。在標準的 3x3 魔術方塊中，共有 6 個中心塊。**

4. **底層十字** ：指在魔術方塊的第一層（通常選用**白色**作為底面），將 **4 個邊塊對齊** 歸位，同時與側邊四面中心塊顏色對齊，使其在底部形成一個「十字型」的狀態。

(二)、**計量標準 ：**

本研究採用 HTM (Half Turn Metric) 作為步數計算標準，即轉動任意層 90 度或 180 度皆視為 1 步。在魔術方塊領域中，HTM 也被稱為 FTM(Face Turn Metric)。

這是世界魔方協會 (WCA) 採用的官方計步標準，其核心規則如下：

1. 定義：任何一個外層轉動，不論旋轉角度是 90 度還是 180 度，都只算作 1 步。
2. 中層轉動 (Slice moves)：在 HTM 規則下，一個中層轉動會被視為兩個外層轉動的組合，因此算作 2 步。
3. 整體旋轉 (Rotations)：旋轉整個方塊，不改變方塊狀態，因此算作 0 步。
4. 應用場景：最常用於計算最少步 (FMC) 比賽中。

表格 1 常見計步方式對比

縮寫	全稱	180 度轉動(U2)	中層(M)
HTM	Half Turn Metric	1 步	2 步
QTM	Quarter Turn Metric	2 步	2 步
STM	Slice Turn Metric	1 步	1 步
ETM	Execution Turn Metric	1 步	視手法而定

(三)、背景介紹：

1. LBL 法 (Layer By Layer)

發展背景：化繁為簡的啟蒙

LBL，即「層先法」，是魔術方塊界流傳最久的基礎解法。

起源： 1980 年代初期，隨著魔術方塊在全球爆紅，最早的一批玩家（如 David Singmaster）開始尋求有系統的解法。當時的思路非常直觀：既然方塊是立體的，那就由底至頂，一層一層解開。

核心邏輯：

(1). 第一層（Bottom Layer）： 先做十字，再解角塊。

(2). 第二層（Middle Layer）： 處理中間層的四個邊塊。

(3). 第三層（Top Layer）： 這是最難的部分，通常拆解成「頂面十字 → 頂面顏色 → 頂層角塊位置 → 頂層邊塊位置」。

歷史意義： 它是目前全世界絕大多數玩家的入門磚。雖然步驟多（約 100 步以上），但記憶量極小（僅需 5-7 個公式），讓魔術方塊從天才的玩具變成了大眾都能學會的遊戲。

2. CFOP 法 (Fridrich Method)

發展背景：為了速度而生的工業標準

雖然 CFOP 的許多概念是集體智慧的結晶，但捷克科學家 Jessica Fridrich 在 1997 年將其系統化並發佈在網路上，因此也被稱為「Fridrich Method」。

(1). C - Cross (底面十字)： 與 LBL 相同，但競速玩家會要求在 8 步內完成，且通常在底部操作以方便觀察。

(2). F - F2L (First Two Layers)： 這是 CFOP 的靈魂。它將 LBL 的第一層角塊和第二層邊塊「合併」處理，一次塞入一對（Pair）。

(3). O - OLL (Orientation of Last Layer)： 用一個公式將頂面顏色全部翻轉一致（共 57 種情況）。

(4). P - PLL (Permutation of Last Layer)： 用一個公式調整頂層方塊的位置（共 21 種情況）。

為什麼 CFOP 會統治競速界？

(1). 減少步數： F2L 大幅縮減了冗餘動作。

(2). 減少觀察次數： 將第三層簡化為兩個步驟（OLL/PLL），讓大腦反應能跟上手速。

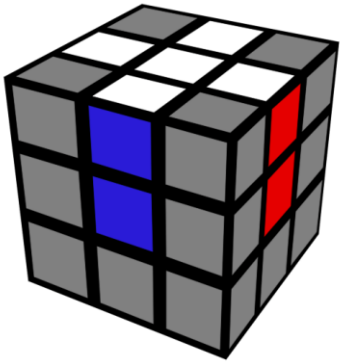
(3). 硬體進化： 隨著「速解方塊」在 2000 年代後的機械改良（如磁力、切角性能），CFOP 這種高連貫性的演算法更能發揮威力。

有趣的小知識： 雖然 Jessica Fridrich 名氣最大，但 F2L 的概念最早可追溯到 1980 年代初期由 René Schoof 提出。Fridrich 的貢獻在於她利用電腦程式算出了最優化的公式集，並無私地分享給全世界，讓更多人一同進入魔方速解的領域。

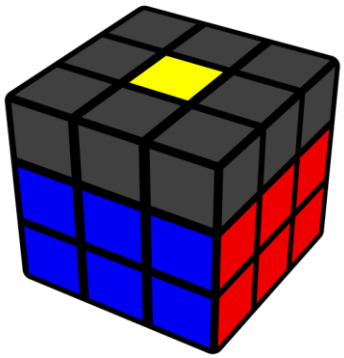
二. 系統流程分析

(一)、CFOP 流程：

1. Cross： 完成底層十字，歸位 4 個底層邊塊，對齊側面中心。



2. F2L (First 2 Layers)： 同時完成底層角塊與中層邊塊，共 4 組配對(pair)。每一組配對都是由一個角塊和一個邊塊組成的 $2\times1\times1$ 方塊，在經過頂層的調整後，總共有 41 條不同的公式。



3. OLL (Orientation of Last Layer)： 翻轉頂層塊體朝向，使頂面顏色統一，也就是完成頂面導向（一共有 57 個公式）。

在魔術方塊的 CFOP 解法中，第三步 OLL (Orientation of the Last Layer) 共有 57 種情況，這是由頂層方塊的幾何排列與旋轉對稱性共同決定的。 以下是為什麼剛好是 57 的數學與邏輯原因：

(1). 理論上的最大組合數

頂層包含 4 個邊塊和 4 個角塊，理論上的最大組合數

i. 邊塊（Edges）：每個邊塊有 2 個方向。4 個邊塊共有 $2^4 = 16$ 種狀態。

ii. 角塊（Corners）：每個角塊有 3 個方向。4 個角塊共有 $3^4 = 81$ 種狀態。

iii. 總初步組合：16×81＝1296 種。

(2). 約束性：

根據魔術方塊的構造，所有的方塊並不是完全獨立。

i. 邊塊 (Edges)：每個邊塊有 2 種朝向。根據魔術方塊的約束性 (Parity)，邊塊翻轉的總數必須是偶數。因此，前 3 個邊塊可以隨機朝向，最後一個邊塊的朝向會被固定，總共有 8 種 邊塊朝向組合。

ii. 角塊 (Corners)：每個角塊有 3 種旋轉角度。同樣地，最後一個角塊的朝向受限於前三個，因此總共有 27 種 角塊朝向組合。

iii. 將兩者相乘：事實上可以存在的可能為 216 種可能的狀態。

(3). 旋轉對稱性

這 216 種狀態中，有很多情況在「視覺上」其實是同一個公式可以解決的，只是方向不同（轉一下 U 層就一樣了）。 在數學上，我們利用波利亞計數定理（Pólya enumeration theorem）或燒錄引理（Burnside's Lemma）來處理這種旋轉對稱性。這 216 種狀態中，許多情況在旋轉頂層（U、U2、U'）後其實是同一個案例。

例如，一個「小 L 型」朝向左上與朝向右下，在邏輯上只需要同一個公式就能解決。

數學上將這些狀態進行對稱分類（考慮 4 個旋轉方向），並扣除完全解好的「Skip」狀態後，最終剩餘 57 個不重複的特徵案例。

(4). 最終的分類統計

經過對稱性歸類後，這 216 個有效狀態被劃分為 58 個情況。1 種情況是「已完成」狀態： 即頂面已經全為同一個顏色，不需要公式。剩下的 57 種情況： 這就是我們通稱的 57 OLL 公式。

(5). 常見分類

為了方便記憶，這 57 種可以進一步細分為：

i. 點型（Dots）： 8 種（頂層邊塊皆未翻正）。

ii. 線型（Lines）： 13 種（頂層兩個邊塊成一直線）。

iii. L 型（L-shapes）： 29 種（頂層兩個邊塊成 L 型）。

iv. 十字型（Crosses）： 7 種（頂層邊塊已完成十字，僅需翻轉角塊）。

v. 跳過 OLL(Skip)： 1 種。

(6). 機率：

i. 完全不對稱的情況： 一種樣式經過 90 度, 180 度, 270 度 旋轉會產生 4 種不同的視覺方向。這類 OLL 出現的機率是 $\frac{4}{216}=\frac{1}{54}$ 。

ii. 具備對稱性的情況： 旋轉 180 度 後長得一模一樣（例如一字型十字）。這類 OLL 出現的機率是 $\frac{2}{216}=\frac{1}{108}$ 。

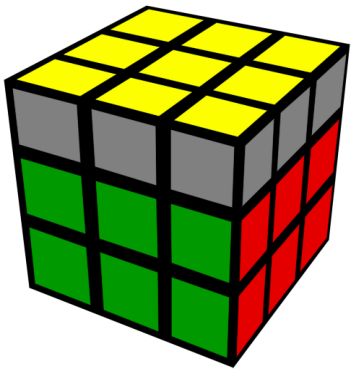
iii. 高度對稱的情況： 旋轉 90 度後都一樣（例如點型十字）。這類機率更低，只有 $\frac{1}{216}$ 。

iv. 並非所有 OLL 都是平等的。L 型 佔了將近一半的出現率（44%），而 點型 雖然只有 12.5%，但是步數卻非常多，是影響速度的關鍵。

(7). 每種分類出現的機率與所需要的步數

表格 2

分類	出現機率	平均步數
點型（Dots）	13.80%	11.5
L 型(L-Shape)	50.00%	10.0
線型（Lines）	22.41%	9.8
十字型（Crosses）	12.07%	7.8
跳過 (OLL Skip)	1.72%	0



4. **PLL (Permutation of Last Layer)**： 頂層排列（21 個公式）。

在魔術方塊的 CFOP 解法中，最後一步 PLL (Permutation of the Last Layer) 共有 21 種情況。這同樣是由頂層方塊的排列組合與旋轉對稱性決定的，但與 OLL 不同的是，PLL 只處理「位置」而不改變「朝向」。

以下是 21 種情況的由來：

(1). 頂層方塊的排列數

在 PLL 階段，所有頂層方塊的顏色都已經朝上（由 OLL 完成），我們只需要交換 4 個邊塊和 4 個角塊的位置。

邊塊排列：4 個邊塊有 4!=24 種排列。

角塊排列：4 個角塊有 4!=24 種排列。

總組合： $24\times24=576$ 種狀態。

(2). 物理約束與對稱性簡化

在魔術方塊的物理結構下，這 576 種狀態會被大幅縮減：

奇偶性約束 (Parity)：在不拆開方塊的情況下，邊塊的排列奇偶性必須與角塊相同。這直接將組合數砍半，剩下 288 種 合法狀態。

旋轉對稱性 (AUF)：由於頂層可以自由旋轉（U、U2、U'），許多狀態在旋轉後本質上是相同的案例。將 288 種狀態除以 4 個旋轉方向，並扣除完全解好的「Solved」狀態，最終得到 72 種 不同的排列。

(3). 鏡像與逆向合併

在 Speedsolving 規則 中，我們通常將鏡像案例（如 Ja 與 Jb）和逆向案例（如 Ua 與 Ub）視為獨立的公式來學習，以追求最快的反應速度。

如果不計鏡像與逆向，實際上只有 13 種 基礎排列模式。但為了在速解時能「一看到就轉」，標準教材將鏡像與逆向拆開，形成了大家熟悉的 21 個 PLL 公式。

(4). 常見案例分類

純邊塊交換 (4 種)：Ua, Ub, H, Z

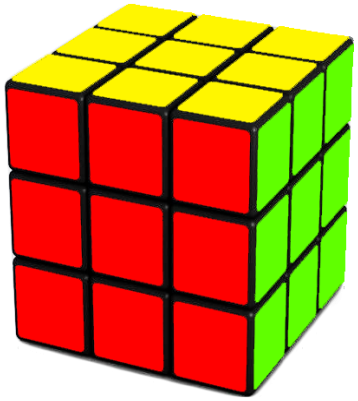
純角塊交換 (3 種)：Aa, Ab, E

邊角同時交換 (14 種)：T, F, Ja, Jb, Ra, Rb, V, Y, Na, Nb, Ga, Gb, Gc, Gd

(5). 每種分類出現的機率與所需要的步數

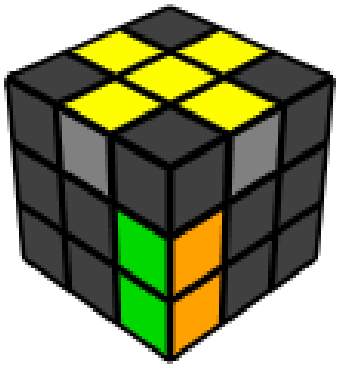
表格 3

分類	出現機率	平均步數
純邊塊交換	19.04%	9~10
純角塊交換	14.29%	9~10
邊角同時交換	66.67%	12~13



(二)、ZBLL 流程（配合 ZBLS/VHLS）：

1. Cross & F2L-1：指的是完成前兩層（共 4 組）中的 前 3 組，留下最後一個「空槽」（Slot），用來執行接下來的 ZBLS（Zborowski-Bruchem Last Slot）。
2. Last Slot + EO(ZBLS)： 最後一組 F2L 同時調整頂層邊塊方向變成十字（Edge Orientation）。



3. ZBLL： 完成頂層所有角塊方向與邊角排列（493 個公式，一步完成）。

在魔術方塊 CFOP 解法的高階領域中，ZBLL (Zborowski-Bruchem Last Layer) 是指在頂層邊塊朝向已正確（即頂層有十字）的情況下，一次完成「角塊朝向」與「所有方塊排列」的技巧。

之所以會有 493 種情況，是基於以下數學推算：

(1). **核心組合計算**

ZBLL 處理的是除了頂層邊塊朝向以外的所有變數：

角塊朝向 (Corner Orientation)：邊塊已好，剩下的角塊朝向有 種組合。

扣除已解好的 1 種，剩下 26 種。這 26 種狀態被歸類為 7 個基礎 OLL 類型（S, AS, Pi, H, U, T, L）。

頂層位置排列 (Permutation)：在邊塊朝向正確的前提下，頂層 8 個方塊（4 角 4 邊）的合法排列數為 46656 種。

(2). **扣除旋轉對稱性**

將「角塊朝向」與「排列」結合，總狀態數非常驚人。但透過數學的群論計算，並扣除旋轉對稱（頂層轉 90/180/270 度後相同的案例），最終得到 493 個獨特案例：

這 493 個案例包含了所有邊塊朝向正確的 PLL (21 種)。

其餘則是將角塊旋轉與位置交換合併處理的情況。

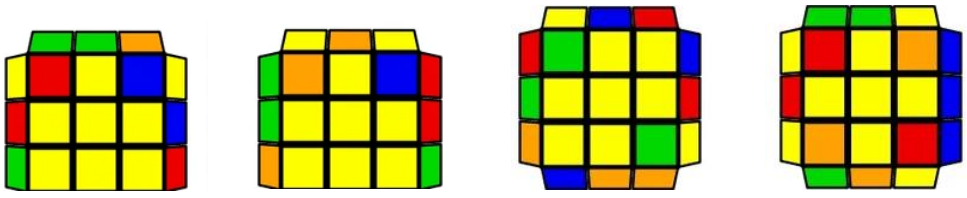
(3). **為什麼要學這麼多？**

一般的做法是 OLL (57) + PLL (21)。而 ZBLL 的目標是把原本需要兩步的「十字型 OLL」與「PLL」合併成一步。

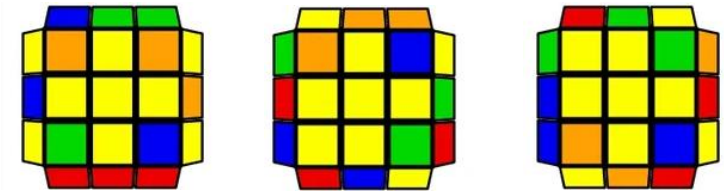
如果學會了這 493 個公式，當你完成 F2L 且頂層有十字時，就能 100% 一步還原 整個頂層。

案例分類（依角塊朝向分組）：

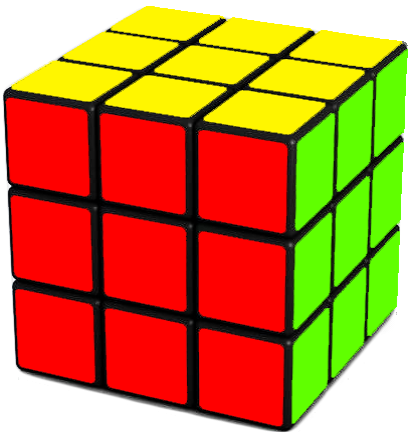
T 組: 72 種　　U 組：72 種　　L 組: 72 種　　Pi 組: 72 種



H 組: 40 種　　S 組: 72 種　　AS 組: 72 種



加上 1 個已解好的狀態(跳 O)，總計為 494 種，但通常通稱為 493 個公式。



三. **數據收集與計算方式**

我們利用窮舉法及加權平均法來計算各階段的期望步數。公式如下 ：

$$\sum S \times P = Savg$$

其中 P 為該情況出現的機率(權重)，S 為該公式的最佳解步數。

而 Savg 是每一階段平均之步數。

透過電腦程式，我們可以快速的計算每個速解流程的加權平均時間。

四. **比較**

最後比較使用 OLL+PLL 和使用 ZBLL 的步數差異為研究結果。

肆、研究結果

一. **CFOP 公式中，57 個不重複的 OLL 加上 PLL 的 21 種情況，頂層共需要 78 個公式。**

二. **CFOP 頂層步數分析期望值 (Standard OLL + PLL)**

在完成 F2L 後，CFOP 需經過 OLL 與 PLL。

(一)、OLL 階段：

1. 標準 57 個 OLL 情況和跳 O。
2. 利用加權平均法算出的平均步數約為 9.7 步 (HTM)。

(二)、PLL 階段：

1. 標準 21 個 PLL 情況和跳 P。
2. 利用加權平均法算出的平均步數約為 11.8 步 (HTM)。

(三)、CFOP 頂層總和：

1. 9.7+ 11.8= 21.5 步。
2. 考慮判斷後的銜接（AUF, Adjust Upper Face），通常需加 0-2 步，總計約 23 步。

三. ZBLL 步數分析期望值

在完成 F2L 後，ZBLL 需經過 EO 與 ZBLL。

(一)、EO(ZBF2L)：

1. 標準 395 個情況。
2. 利用加權平均法算出的平均步數約為 9.5 步 (HTM)。

(二)、ZBLL 階段：

1. 標準 472 個 ZBLL 情況和 21 個 PLL，共 493 個公式。
2. 利用加權平均法算出的平均步數約為 14.5 步 (HTM)。

(三)、ZBLL 頂層總和：

1. 9.5+ 14.5= 24 步。
2. 考慮判斷 ZBLL 後不需要銜接，所以不用加上調整的步數。

ZBLL 的發動條件是「頂層邊塊方向(EO)已正確」（十字已好）。

四. ZBLL 所需的邊塊導向前置動作（EO）對總步數的影響。

表格 4

比較項目	標準 CFOP	進階 ZBLL 系統	差異 (ZBLL - CFOP)
Last Slot (LS，前置作業)	7 步	9.5 步 (含 EO)	+2.5 步
頂層處理 (LL)	21.5 步 (OLL+PLL)	14.5 步 (ZBLL)	-7.0 步
總計差異	**基準**	**省步**	**淨節省約 4.5 步**

五. 兩種系統的理論最小步數與實戰應用差異。

ZBLL 實際淨效益約為 4.5 步。在速解中，頂尖選手的手速約 10 TPS（每秒 10 步），4.5 步理論上可節省 0.45 秒。

六. ZBLL 和 CFOP 頂層反應次數與時間的比較。

(一)、**頂層反應時間數據對比 (平均值)**

表格 5

階段	OLL + PLL (傳統)	ZBLL (進階)
第一次觀察	0.3s (OLL 圖形極易辨識)	0.5s - 0.8s (需判斷角塊位置與側面關係)
公式間銜接	0.2s - 0.4s (等待頂層轉動以觀察 PLL)	**無**
第二次觀察	0.3s (PLL 特徵明顯)	**無**
總觀察時間	**0.8s - 1.0s**	**0.5s - 0.8s**

(二)、**關鍵差異分析**

1. **反應時間：**

(1). **OLL+PLL：**OLL 只看頂面顏色，PLL 只看側面色塊關係。大腦處理壓力小，反應極快。

(2). **ZBLL：**必須同時看**頂面方向**、**角塊排列 (CP)** 以及**稜塊位置 (EP)**。這導致單次反應時間通常比單次 PLL 慢約 **0.2s - 0.3s**。

2. **「轉動觀察」(AUF) 的代價：**

(1). OLL 結束後，通常需要做一次 U 層轉動（AUF）來確認 PLL 圖形，這個動作會產生 **0.2s 左右的視覺延遲**。

(2). ZBLL 省去了這個中間環節，一旦公式開始執行，中途不會有任何停頓。

伍、討論

一. 步數效益分析

經由電腦實作的數據顯示，雖然 ZBLL 單獨看能節省近 7 步，但扣除為了「做球」（Setup EO）所多花的步數，實際淨效益約為 4.5 步。在速解中，頂尖選手的手速約 10 TPS（每秒 10 步），4.5 步理論上可節省 0.45 秒。

二. 判斷時間

CFOP 的優勢在於 OLL 和 PLL 的圖形辨識相對簡單且特徵明顯。ZBLL 則需要在邊塊導向正確後，同時判斷角塊方向與所有塊的排列位置，其辨識難度呈指數上升。若選手的判斷時間增加超過 0.5 秒，則會抵銷步數減少帶來的時間優勢。且為了確保 EO 導向正確，多數選手還需額外記憶 305 條 ZBLS 的公式。

三. 機率問題

在標準 CFOP 中，有 1/8 的機率在 F2L 完成後邊塊自然導向正確（Skip EO）。在這種情況下，若選手掌握 ZBLL，則完全不需要額外的 Setup 成本，直接享受 7 步的縮減優勢。因此，「部分 ZBLL」（如 ZBLL 的 T, U, L 集合） 可能是比全套 ZBLL 更具投資報酬率的策略。

四. 頂層反應分析

ZBLL 需要觀察較多排列，但 OLL 和 PLL 一共需要兩次觀察時間，ZBLL 只需要一次觀察，一旦公式開始執行，中途不會有任何停頓，所以 ZBLL 觀察時間會比 CFOP 還快約 0.2 秒。頂尖選手在步數上可節省約 0.5 秒，在反應時間上可節省約 0.2 秒，所以 ZBLL 會較 CFOP 快出 0.7 秒。

陸、結論

一. **步數絕對優勢：** 在統計與期望值計算下，ZBLL 系統相較於傳統 CFOP 系統，平均能減少約 4.5 步 (HTM) 的還原路徑。

二. **條件限制：** 此優勢建立在熟練掌握 493 個公式以及能快速完成邊塊導向（EO）的前提下，因此公式熟練及靈活的應用是非常重要的。而為了在頂層階段有正確之邊塊導向，大部分選手還必須搭配 ZBLS 305 條公式，總共有將近 800 條，因此需要龐大的時間成本。

三. **建議策略：** 對於初學者與進階者，CFOP 仍是最佳入門選擇；它公式較少，相對容易學習。對於追求極限（Sub-8 秒）的頂尖選手，學習 ZBLL 是突破秒數瓶頸的有效數學手段。

四. **未來展望：** 未來可進一步研究「公式數量」對總還原時間和反應時間的影響，以及考慮物理手感、轉體等因素，找出 ZBLL 的「黃金損益平衡點」。

柒、參考文獻

1. idrich, J. (1982). *The Fridrich Method*.
2. Speedsolving.com Wiki. (n.d.). *ZBLL Statistics and Probability*.
3. Kociemba, H. (1992). *The Two-Phase Algorithm*.
4. Feliks Zemdegs. (2018). *Cubeskills - Advanced ZBLL tutorials*.
5. CubeRoot ZBLL https://cuberoot.me/wp-content/uploads/2019/10/167-ZBLL.pdf
6. Speed Cube Database ZBLLhttps://www.speedcubedb.com/a/ZBLL

附錄

1.ZBLL 公式表

https://www.scribd.com/document/552055233/Anthony-Brooks-ZBLL

2.打亂輔助程式(由於加權程式過長，故不附於此)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <map>
#include <random>
#include <ctime>
#include <sstream>

using namespace std;

// 定義魔方的六種顏色
enum Color {
    WHITE, YELLOW, GREEN, BLUE, RED, ORANGE
};

// 顏色到字元的映射 (方便列印)
const map<Color, char> color_map = {
    {WHITE, 'W'}, {YELLOW, 'Y'}, // 白, 黃
    {GREEN, 'G'}, {BLUE, 'B'},   // 綠, 藍
    {RED, 'R'}, {ORANGE, 'O'}    // 紅, 橘
};
```

```
// 每個面是 3x3 的顏色網格
using Face = vector<vector<Color>>;

class Cube {
private:
    // 儲存六個面 (U:上, D:下, F:前, B:後, L:左, R:右)
    Face U, D, F, B, L, R;

    // 輔助函數：執行 3x3 矩陣的順時針旋轉 (用於面轉動)
    Face rotate_face_clockwise(const Face& face) {
        Face new_face(3, vector<Color>(3));
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                // (i, j) 轉到 (j, 2-i)
                new_face[j][2 - i] = face[i][j];
            }
        }
        return new_face;
    }

    // 輔助函數：根據轉動字串返回其軸線 (R/L -> 'X', U/D -> 'Y', F/B -> 'Z')
    char get_axis(const string& move_str) const {
        char base_move = move_str[0];
        if (base_move == 'R' || base_move == 'L') return 'X';
        if (base_move == 'U' || base_move == 'D') return 'Y';
        if (base_move == 'F' || base_move == 'B') return 'Z';
        return ' ';
    }

    // 輔助函數：在執行完動作後列印當前狀態
    void print_step(const string& move_str) const {
```

```
        cout << "\n--- 執行動作: " << move_str << " ---\n";


        // 簡化列印，只印出每個面中心的顏色和部分邊緣

        cout << "中心顏色: U(" << color_map.at(U[1][1]) << ") D(" << color_map.at(D[1][1]) << ") F(" << color_map.at(F[1][1]) << ") B(" << color_map.at(B[1][1]) << ") L(" << color_map.at(L[1][1]) << ") R(" << color_map.at(R[1][1]) << ")\n";


        // 詳細展開圖

        print_cube();

    }



public:

    // 構造函數：初始化一個已解好的魔方狀態

    Cube() {

        U = Face(3, vector<Color>(3, YELLOW));  // 上面

        D = Face(3, vector<Color>(3, WHITE));   // 下面

        F = Face(3, vector<Color>(3, GREEN));   // 前面

        B = Face(3, vector<Color>(3, BLUE));    // 後面

        L = Face(3, vector<Color>(3, RED));     // 左面

        R = Face(3, vector<Color>(3, ORANGE));  // 右面

    }


    // 顯示魔方狀態 (展開圖形式)

    void print_cube() const {


        // 顯示 U 面 (上面，置中)

        cout << "      U (Y)\n";

        for (int i = 0; i < 3; ++i) {

            cout << "      ";

            for (int j = 0; j < 3; ++j) {
```

```
        cout << color_map.at(U[i][j]) << " ";
    }
    cout << "\n";
}

// 顯示 L, F, R, B 四面 (中間一排)
cout << "  L(R)F(G)R(O)B(B)\n";
for (int i = 0; i < 3; ++i) {
    // L
    for (int j = 0; j < 3; ++j) { cout << color_map.at(L[i][j]) << " "; }
    cout << " ";
    // F
    for (int j = 0; j < 3; ++j) { cout << color_map.at(F[i][j]) << " "; }
    cout << " ";
    // R
    for (int j = 0; j < 3; ++j) { cout << color_map.at(R[i][j]) << " "; }
    cout << " ";
    // B
    for (int j = 0; j < 3; ++j) { cout << color_map.at(B[i][j]) << " "; }
    cout << "\n";
}

// 顯示 D 面 (下面，置中)
cout << "      D (W)\n";
for (int i = 0; i < 3; ++i) {
    cout << "      ";
    for (int j = 0; j < 3; ++j) {
        cout << color_map.at(D[i][j]) << " ";
    }
    cout << "\n";
}
```

```
    cout << "\n";
}

// --- 順時針轉動 (Clockwise Moves) ---
void move_R() {
    R = rotate_face_clockwise(R);
    vector<Color> temp_col;
    for (int i = 0; i < 3; ++i) { temp_col.push_back(U[i][2]); }
    for (int i = 0; i < 3; ++i) { U[i][2] = F[i][2]; }
    for (int i = 0; i < 3; ++i) { F[i][2] = D[i][2]; }
    for (int i = 0; i < 3; ++i) { D[i][2] = B[2 - i][0]; }
    for (int i = 0; i < 3; ++i) { B[2 - i][0] = temp_col[i]; }
}

void move_L() {
    L = rotate_face_clockwise(L);
    vector<Color> temp_col;
    for (int i = 0; i < 3; ++i) { temp_col.push_back(U[i][0]); }
    for (int i = 0; i < 3; ++i) { U[i][0] = B[2 - i][2]; }
    for (int i = 0; i < 3; ++i) { B[2 - i][2] = D[i][0]; }
    for (int i = 0; i < 3; ++i) { D[i][0] = F[i][0]; }
    for (int i = 0; i < 3; ++i) { F[i][0] = temp_col[i]; }
}

void move_U() {
    U = rotate_face_clockwise(U);
    vector<Color> temp_row;
    for (int i = 0; i < 3; ++i) { temp_row.push_back(F[0][i]); } // 暫存 F[0]
    // F <- R(正向): R[0][i] -> F[0][i]
    for (int i = 0; i < 3; ++i) { F[0][i] = R[0][i]; }
    // R <- B(正向): B[0][i] -> R[0][i]
```

```
    for (int i = 0; i < 3; ++i) { R[0][i] = B[0][i]; }
    // B <- L(正向): L[0][i] -> B[0][i]
    for (int i = 0; i < 3; ++i) { B[0][i] = L[0][i]; }
    // L <- F_temp(正向): temp_row[i] -> L[0][i]
    for (int i = 0; i < 3; ++i) { L[0][i] = temp_row[i]; }
}

void move_D() {
    D = rotate_face_clockwise(D);
    vector<Color> temp_row;
    for (int i = 0; i < 3; ++i) { temp_row.push_back(F[2][i]); } // 暫存 F[2]
    // F <- L(正向): L[2][i] -> F[2][i]
    for (int i = 0; i < 3; ++i) { F[2][i] = L[2][i]; }
    // L <- B(正向): B[2][i] -> L[2][i]
    for (int i = 0; i < 3; ++i) { L[2][i] = B[2][i]; }
    // B <- R(正向): R[2][i] -> B[2][i]
    for (int i = 0; i < 3; ++i) { B[2][i] = R[2][i]; }
    // R <- F_temp(正向): temp_row[i] -> R[2][i]
    for (int i = 0; i < 3; ++i) { R[2][i] = temp_row[i]; }
}

// **F 轉動 (F Clockwise)**: U -> R -> D -> L -> U
void move_F() {
    F = rotate_face_clockwise(F);
    vector<Color> temp_row;
    for (int i = 0; i < 3; ++i) { temp_row.push_back(U[2][i]); } // 暫存 U[2]

    // U[2] <- L 右邊列 (反向)
    for (int i = 0; i < 3; ++i) { U[2][i] = L[2 - i][2]; }
```

```
    // L 右邊列 <- D 頂排 (正向)
    for (int i = 0; i < 3; ++i) { L[i][2] = D[0][i]; }

    // D 頂排 <- R 左邊列 (反向)
    for (int i = 0; i < 3; ++i) { D[0][i] = R[2 - i][0]; }

    // R 左邊列 <- U 底排 (正向)
    for (int i = 0; i < 3; ++i) { R[i][0] = temp_row[i]; }


}

// **B 轉動 (B Clockwise)**: U <- R -> D <- L -> U (含必要反序)
void move_B() {
    B = rotate_face_clockwise(B);
    // 暫存 U 的上排
    vector<Color> temp_row = U[0];

    // U 的上排 <- R 的右列（正向）
    for (int i = 0; i < 3; ++i) {
        U[0][i] = R[i][2];
    }
    // R 的右列 <- D 的下排（反序）
    for (int i = 0; i < 3; ++i) {
        R[i][2] = D[2][2 - i];
    }
    // D 的下排 <- L 的左列（正向）
    for (int i = 0; i < 3; ++i) {
        D[2][i] = L[i][0];
    }
```

```
    // L 的左列 <- 暫存的 U 上排（反序）
    for (int i = 0; i < 3; ++i) {
        L[i][0] = temp_row[2 - i];
    }
}

// --- 逆時針轉動 (Prime Moves) ---
void move_R_prime() { move_R(); move_R(); move_R(); }
void move_L_prime() { move_L(); move_L(); move_L(); }
void move_U_prime() { move_U(); move_U(); move_U(); }
void move_D_prime() { move_D(); move_D(); move_D(); }
void move_F_prime() { move_F(); move_F(); move_F(); }
void move_B_prime() { move_B(); move_B(); move_B(); }

// --- 轉動執行器 (Move Executor) ---
void execute_move(const string& move_str) {
    if (move_str == "R") move_R();
    else if (move_str == "R'") move_R_prime();
    else if (move_str == "R2") { move_R(); move_R(); }
    else if (move_str == "L") move_L();
    else if (move_str == "L'") move_L_prime();
    else if (move_str == "L2") { move_L(); move_L(); }
    else if (move_str == "U") move_U();
    else if (move_str == "U'") move_U_prime();
    else if (move_str == "U2") { move_U(); move_U(); }
    else if (move_str == "D") move_D();
    else if (move_str == "D'") move_D_prime();
    else if (move_str == "D2") { move_D(); move_D(); }
    else if (move_str == "F") move_F();
    else if (move_str == "F'") move_F_prime();
    else if (move_str == "F2") { move_F(); move_F(); }
```

```
    else if (move_str == "B") move_B();
    else if (move_str == "B'") move_B_prime();
    else if (move_str == "B2") { move_B(); move_B(); }
    else {
        cerr << "錯誤：無法識別的轉動指令 (" << move_str << ")\n";
    }
}

// --- 序列執行器 (Sequence Executor) ---
// print_steps 設為 true 時，將在每一步後列印狀態。
void execute_sequence(const string& sequence, bool print_steps = false) {
    stringstream ss(sequence);
    string move;
    int step = 0;

    cout << "\n--- 開始執行序列 ---\n";
    while (ss >> move) {
        step++;
        if (print_steps) {
            cout << "步驟 " << step << ": ";
        }
        execute_move(move);
        if (print_steps) {
            print_step(move);
        }
    }
    cout << "--- 序列執行完畢 ---\n";
}

// --- 打亂程式 (Scramble Generator) ---
```

```
string scramble(int length = 25, bool print_steps = false) {
    const vector<string> moves = {
        "R", "R'", "R2", "L", "L'", "L2",
        "U", "U'", "U2", "D", "D'", "D2",
        "F", "F'", "F2", "B", "B'", "B2"
    };

    static default_random_engine generator(time(0));
    uniform_int_distribution<int> distribution(0, moves.size() - 1);

    stringstream scramble_sequence;
    char last_axis = ' ';
    int step = 0;

    cout << "\n=== 打亂公式 ===\n";  // 新增：顯示打亂開始

    for (int i = 0; i < length; ++i) {
        string next_move;
        char next_axis;

        do {
            next_move = moves[distribution(generator)];
            next_axis = get_axis(next_move);
        } while (next_axis == last_axis);

        execute_move(next_move);
        step++;

        // 新增：即時顯示每一步的打亂動作
        cout << next_move << " ";
        // 每 5 步換行，提高可讀性
```

```
            if (step % 5 == 0) cout << "\n";

            if (print_steps) {
                cout << "\n 步驟 " << step << ": ";
                print_step(next_move);
            }

            last_axis = next_axis;
            scramble_sequence << next_move << " ";
        }

        cout << "\n=== 打亂完成 ===\n";  // 新增：顯示打亂結束
        return scramble_sequence.str();
    }
};

int main() {
    srand(time(0));

    // 新增打亂測試
    cout << "\n=== 測試魔方打亂 ===\n";
    Cube scramble_test;

    cout << "初始狀態：\n";
    scramble_test.print_cube();

    // 執行 25 步打亂，print_steps 設為 false 只顯示打亂公式，不顯示每步狀態
    scramble_test.scramble(25, false);

    cout << "\n 打亂後狀態：\n";
```

```
    scramble_test.print_cube();
   return 0;
}
```