Rubik's-Cube Solver Using HDA*

Parallel Programming Final Report

0716206
陳昱丞
National Yang Ming Chiao Tung University
Hsinchu

0716221
余忠旻
National Yang Ming Chiao Tung University
Hsinchu

309553040
葉柏廷
National Yang Ming Chiao Tung University
Hsinchu

ABSTRACT

狀態空間搜尋(State-space search)在電腦科學領域有非常廣泛的應用。搜尋的目標是要在狀態空間中找到近似最佳解[1]。在本篇文章中，我們實作出三種不同的搜尋演算法:BFS(Breadth-First Search)[2], A*[3]及 Iterative deepening A*(IDA*)[4]，並且將上述演算法平行化[5]後以魔術方塊(Rubik's Cube)作為目標任務來比較彼此的效能差異。我們的實驗結果顯示出三種平行化後的演算法都能得到相當好的加速效能。另外，我們原本 proposal 的題目是寫用 PMIDA*這個算法，但後來發現 HDA*更符合我們的需求。故稍作變更。

INTRODUCTION

魔術方塊 (Rubik's Cube)，在中國大陸稱為魔方，在香港稱為扭計骰，為由匈牙利建築學教授暨雕塑家魯比克·厄爾諾(Rubik Ernő) 於 1974 年發明的機械益智玩具，它的目標是將打亂的方塊變成每面都為相同顏色。以隨機打亂的方塊作為起始狀態，我們可以輕易預期以此狀態作一步旋轉後會得到什麼狀態。以上述想法為基礎，將當前狀態視為節點 (Node)，並向外分支到以此狀態轉一步後所形成的節點，便能形成狀態空間圖 (Graph) 或樹 (Tree)，如此以來就能使用各種搜尋演算法來解決魔術方塊問題。

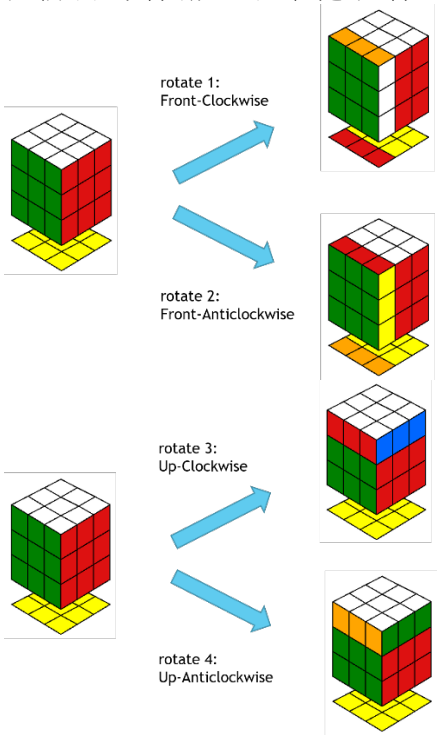
搜尋演算法在許多智力遊戲中都有著相當不錯的表現空間。包括解數獨、解 puzzle problem、或是本篇的解魔方。常見的搜尋演算法有五種，分別是 Breadth-First Search (BFS)、Depth-First Search (DFS)、Iterative Deepening Depth-First Search (IDS)、A star (A*)、Iterative Deepening A star (IDA*)。狀態空間會占用相當大量的記憶體空間，在標準的 3x3x3 魔術方塊中，總共有 4.3252×10^{19} 種狀態，因此需採取其他策略來減少記憶體使用量而不是顯性的觀察下個狀態是否達到目標。屬於啟發式搜尋(heuristic search)的演算法如 A*和 IDA*，能夠估計路徑的好壞來優先選擇搜尋較好的路徑，進一步的提升搜尋效率。

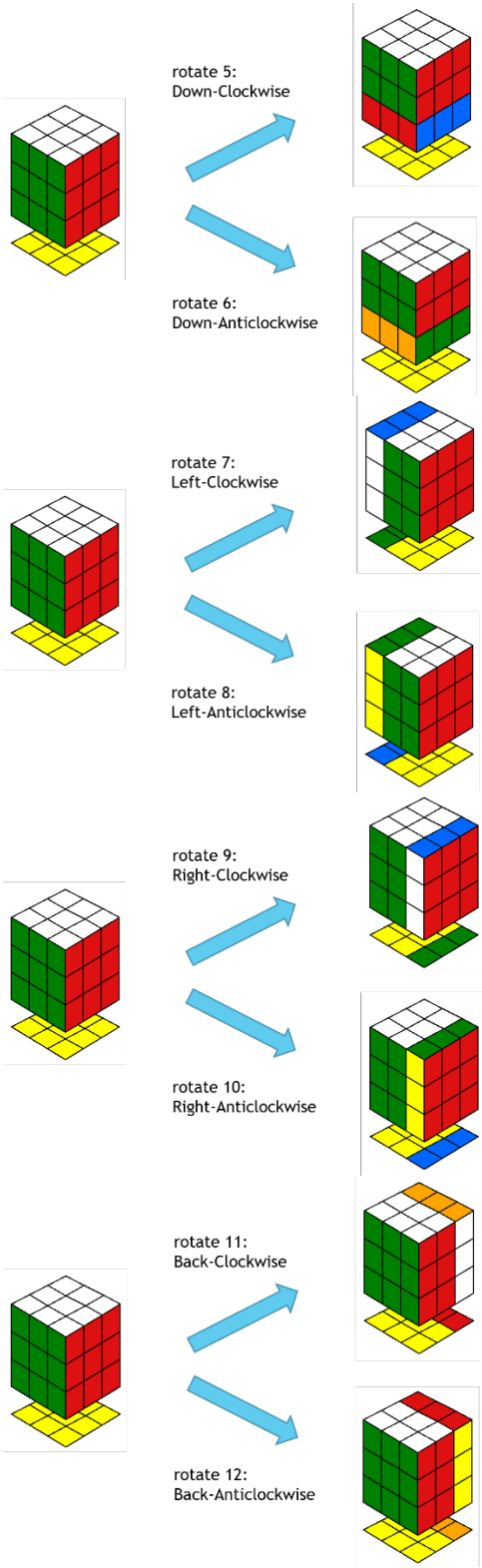
PROPOSED SOLUTION

在 Rubik's solver 中，我們預計使用三種演算法復原被打亂的魔術方塊。一種是暴力法的 BFS、另外兩種是啟發式的 A* 和 IDA*。啟發式的 heuristic function 我們預計採用曼哈頓距離 (Manhattan Distance) 來計算，主要計算色塊之間的距離和來估計其到終點狀態 (terminal state) 之步數。然而，隨著求解魔術方塊的步驟，解的時間、狀態的數量都會呈指數增長。因此我們預計將這三種方法平行化來加速。

我們的算法主要可以定義狀態及旋轉、檢查終止條件、演算主體、平行化四個部分：

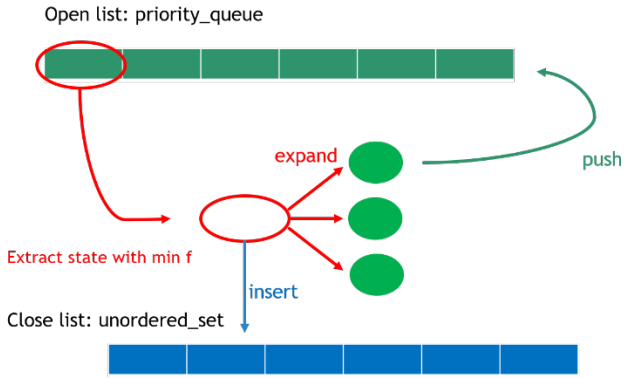
1. 建立初始的魔術方塊儲存狀態以及 transition function，魔術方塊的狀態我們用 18 * 3 陣列儲存，用來表示每個面的各橫列依序的顏色，transition function 執行的動作則是可以分成 12 種，FrontClockwise, FrontAnti-Clockwise, UpClockwise, UpAnti-Clockwise, DownClock-wise, DownAnti-Clockwise, LeftClockwise, LeftAnti-Clockwise, RightClockwise, RightAnti-Clockwise, BackClockwise, BackAnti-Clockwise，用來表示魔術方塊各個面的轉動。以下是各轉法的圖形化：



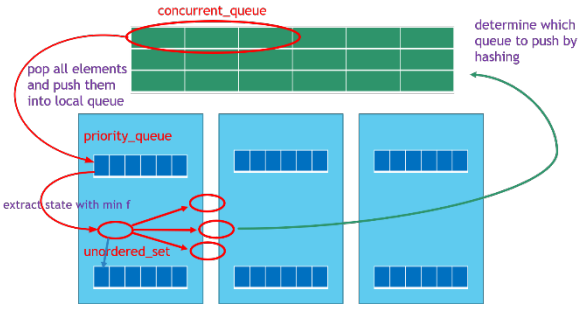


2. 演算過程中當六面顏色變成一致時，也就是到達到了 terminal state。我們可以印出產生的轉法步驟、步數、狀態節點數和所花費的時間，藉此評估每個演算法之效能。

3. 演算主體的部分主要講解 A* State Space Search。A* 需要維護一個 open list 和 close list。在 C++ 中 open list 可用 priority queue 實作、而 close list 用 unordered set。每次都將 priority queue 的 top pop 出來，而 top 值就是 queue 中 f 最小的 node (f = g + h ，g 為當前步數、h 為預估至終點狀態之步數)。接著將該 node 置入 close list 中並進行 expand。Expand 出來的 nodes 再 push 回 priority queue。而 BFS 就是將 priority queue 改成用 queue、IDA*就是逐步放寬 f 的條件限制，也就是多做幾次 A*。上述 A* 流程大致如下圖:



4. 最後是要將上述 3. 的演算法平行化。我們採用 OpenMP 和 Pthread 來實作平行化的部分。在這個 case 中有兩個地方可以平行化。第一個是 Heuristic Calculation 的地方可以使用 OpenMP parallel for reduction 來平行化計算；第二個是套用的平行化算法 Hash Distributed A star (HDA*)[6]。在 hash distributed A star 中，以三個 thread 為例。首先，在 thread 外面會有 3 個 global 的 concurrent queue， 一個 queue 對應一個 thread。而 C++ 中的 tbb :: concurrent queue 就是 thread safety 的 queue， 每個 thread 都可以對它進行 push 或 pop。在每個 thread 中又有自己 local 的 open list 跟 close list。第一步 thread 會先從自己對應的 global queue 中把裡面全部的元素都拿出來並 push 進自己的 local priority queue。 接下來就是做跟上述 A* 一樣的運算。但 expand 出來的 nodes 不是 push 回 priority queue，而是 push 到外面的 global concurrent queue。至於每個 node 要 push 到哪個 queue 由一個 hash function 決定，因而稱作 Hash Distributed。這個架構除了用在 A * 外也可以套用在 BFS 跟 IDA* 上。以下三種套用此架構平行化後的算法我們分別稱之為 HDBFS、HDA*、HDIDA*。以下是 HDA* 大致的流程圖:



另外，我們在這篇當中，不會特別針對 Parallelized Heuristic Calculation 進行討論， Parallelized Heuristic Calculation 雖然能一定幅度提升執行效率，但 HDA* 提升的效果更明顯，並且我們認為同時在 HDA* 的架構下加上 Parallelized Heuristic Calculation ，並不會使 speedup 上升，畢竟 HDA* 已充分利用 CPU 每個 core 的效能，再將 Heuristic Parallelized 是沒辦法達到加速效果，反而有可能因為一些額外的 overhead 而拖慢執行效率。

EXPERIMENT METHODOLOGY

本實驗是以 BFS、A*、IDA* 的演算法為基礎，比較 Rubik's solver 程式的 serial 執行時間和本篇提出的 HDBFS、HDA*、HDIDA* 並加以分析。

我們進行兩種實驗方式，在實驗中，我們採用 6 核心 12 執行緒的測試環境。首先第一種，我們產生 5 至 9 步的測資各一筆，接著分別使用不同的演算法執行同一筆 20 次左右取平均，紀錄時間並比較。由於每差一步執行時間是呈指數成長，相同步數但不同的測資，執行時間可能相差甚多。步數越多此現象則越為明顯。因此我們每一步固定使用一種測資的實驗方式，適合觀察每一步之中，serial 和 parallel 的差距。

而我們第二種實驗測資是採用 random generated，因此每種步數都會是不同的測資取平均。這樣的實驗方式適合觀察步數之間的變化趨勢。

EXPERIMENT RESULTS

以下是第一種實驗之結果: (時間單位: ms)

5 steps			
	BFS	A*	IDA*
Serial	296.049	1413.143	2318.587
Parallel	174.697	345.431	354.497

6 steps			
	BFS	A*	IDA*
Serial	9927.328	3366.185	13333.527
Parallel	4123.196	889.508	2295.892

7 steps			
	BFS	A*	IDA*
Serial	93875.155	51528.373	167112.603
Parallel	32128.963	4729.377	23195.195

8 steps			
	BFS	A*	IDA*
Serial	552603.583	100007.648	270426.444
Parallel	120602.662	6964.098	39615.739

9 steps			
	BFS	A*	IDA*
Serial	>30 min	385544.310	>30 min
Parallel	>30 min	154482.729	>30 min

以下為第二種實驗之結果: (時間單位: ms)

5 steps			
	BFS	A*	IDA*
Serial	498.581	532.645	1184.61
Parallel	530.978	115.363	226.403

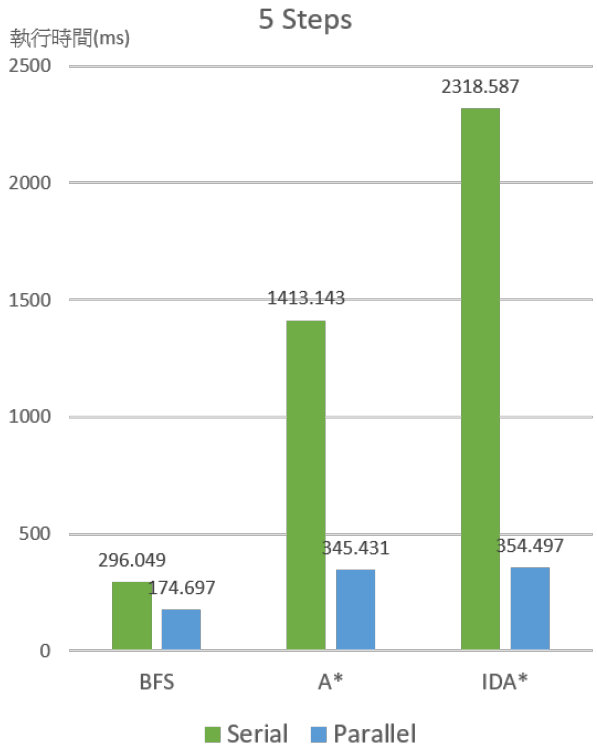
6 steps			
	BFS	A*	IDA*
Serial	6360.88	2817.07	6081.63
Parallel	4660.1	625.258	991.442

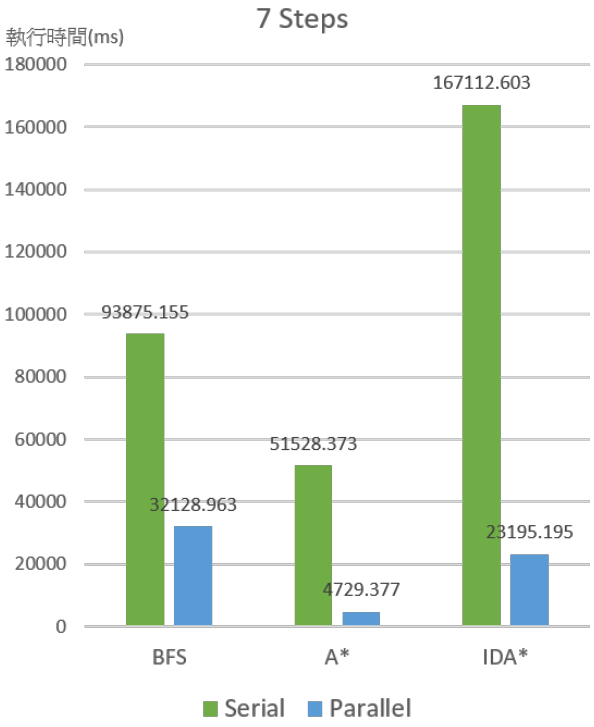
7 steps			
	BFS	A*	IDA*
Serial	58937.9	23598.5	44935.9
Parallel	50233.6	2687.57	4138.73

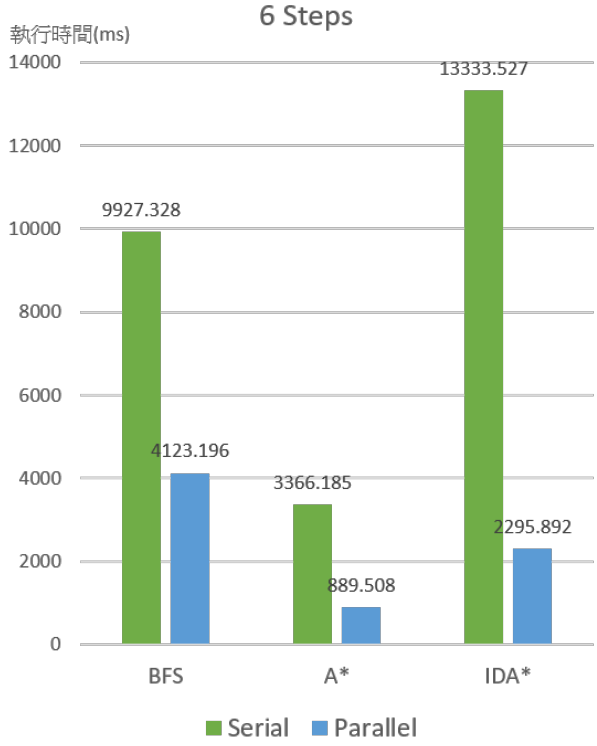
8 steps			
	BFS	A*	IDA*
Serial	390757	156396	351065
Parallel	195467	20839	54681

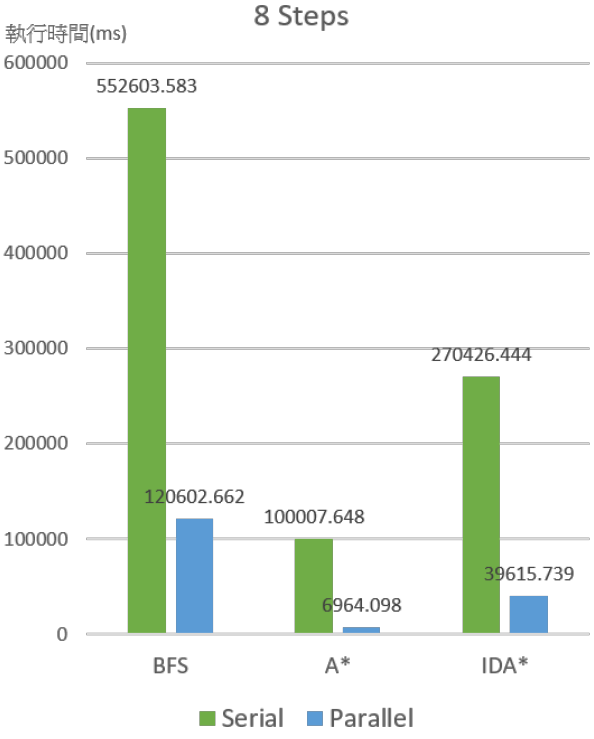
9 steps			
	BFS	A*	IDA*
Serial	>30 min	1829108	>30 min
Parallel	>30 min	347240	792245

實驗一長條圖:

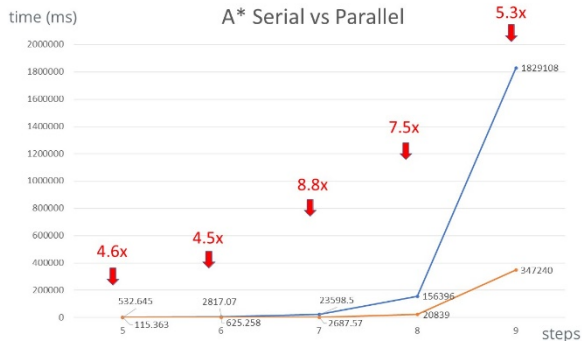

5 Steps
執行時間(ms)
2500
2000
1500
1000
500
0
296.049
174.697
1413.143
345.431
2318.587
354.497
BFS
A*
IDA*
Serial
Parallel


7 Steps
執行時間(ms)
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
93875.155
32128.963
51528.373
4729.377
167112.603
23195.195
BFS
A*
IDA*
Serial
Parallel


6 Steps
執行時間(ms)
14000
12000
10000
8000
6000
4000
2000
0
9927.328
4123.196
3366.185
889.508
13333.527
2295.892
BFS
A*
IDA*
Serial
Parallel


8 Steps
執行時間(ms)
600000
500000
400000
300000
200000
100000
0
552603.583
120602.662
100007.648
6964.098
270426.444
39615.739
BFS
A*
IDA*
Serial
Parallel

實驗二折線圖 (取 A* 部分):



RELATED WORK

搜尋演算法可以分為兩類: Brute-Force Search 和 Heuristic Search[7]。Brute-Force Search 是最泛用的演算法，因為他們不需要任何的領域知識 (Domain Knowledge)，經典的演算法有 BFS、DFS、IDS 等。Heuristic Search 則是需要利用一些先備知識來增進效能，我們在此篇所採用的是 Manhattan distance 的 Heuristic Search 解法。另外，在解魔術方塊中最為知名的是 Pattern Database (Richard E. Korf, 1997)[8]。Pattern Database 的想法是預先計算 IDA* 中使用的 heuristic function (e.g., manhattan distance) 並建立表格，並將狀態建立對應的索引，在計算時直接查詢表格來減少計算時間。

CONCLUSIONS

在分析結果之前，先分析 HDA* 架構的優缺點:

優點	1. 以 global 的 concurrent queue 作為每個 thread 的唯一溝通介面，存取效率高且維護成本低。 2. 每個 thread 擁有自己 local 的 open list 和 close list，運算過程獨立、不互相干擾、不需要同步機制。 3. 以 hash 的方式決定 thread 產生出來的 node 要被 push 進哪個 global queue，也就是決定將來這個 node 將被哪個 thread 執行。若 hash 得好可達成 load balancing。
缺點	1. 每個 thread 擁有自己的 close list 且不互相溝通，可能導致同一個 node 被好幾個 thread 算到，增加許多不必要的運算，拖慢搜尋效率。 2. 若 hash 得不好，可能導致某些 global queue 裡面有很多個 node，而某些 global queue 為空。很多 node 的 queue 所對應到的 thread loading 很重；空的 queue 所對應到的 thread 就只能 busy waiting 等到有其他 thread 將 node push 進來。

從 experiment result 的圖表可以看出，三種演算法平行化後都能得到很好的加速。其中加速效果最明顯的是 HDA*，可以比普通A* 快到 10 倍左右。接著是 HDIDA*，加速效果也有約 6 至 8 倍。最不明顯的是 HDBFS，也有 1.5 至 3 倍的加速。另外，步數越高，平行化加速效果愈加明顯，到 8 步時達到頂峰，甚至可以有超過 10 倍的加速效果。但到了 9 步後，加速效果便有所下降。我們認為是因為 state 數量過大，導致有太多 nodes 被不同 threads 重複計算到，進而拖慢了執行效率。

另外，除了使用 OpenMP 之外，我們也有用 Pthread 來實作 HDA* 這個架構。我們會發現其實執行時間差不多。這是因為 Pthread 的優勢在於他可以提供使用者更詳細的平行化操作。但這個 case 用 Pthread 也只是 create thread 再 join 回去而已，那跟用 OpenMP 基本上沒差。OpenMP 實作起來還更快、更方便。

REFERENCES

[1] Weixiong Zhang. 1999. *State-space Search: Algorithms, Complexity, Extensions, and Applications*. Springer-Verlag New York, Inc.
[2] Wikipedia, 2021. *Breadth-first search*. DOI: https://www.wikiwand.com/en/Breadth-first_search
[3] Wikipedia. 2021. *A* search algorithm*. DOI: https://www.wikiwand.com/en/A*_search_algorithm#
[4] Wikipedia. 2021. *Iterative deepening A**. DOI: https://www.wikiwand.com/en/Iterative_deepening_A*
[5] V. Nageshwara Rao, Vipin Kumar and K. Ramesh. 1987. *A parallel Implementation of Iterative-Deepening-A**. AAAI-87 Proceedings.
[6] Scalable, parallel best-first search for optimal sequential planning DOI: https://dl.acm.org/doi/10.5555/3037223.3037250
[7] Richard E. Korf. 2010. *Artificial Intelligence Search Algorithms*. Algorithms and Theory of Computation Handbook: Special Topics and Techniques, Chapman & Hall/CRC.
[8] Richard E. Korf. 1997. *Finding Optimal Solutions to Rubik's cube Using Pattern Databases*. AAAI DOI: https://dl.acm.org/doi/10.5555/1867406.1867515