ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

Boosting Bidirectional A* Efficiencies: State-nonexistence Fast-confirming Hashing Schemes and Partial Problem-based Informed Heuristic Generations

Kee-cheol Lee

Computer Engineering Department, Hongik University, Seoul, Korea 121-791

Abstract: It is well-known that most games and real world problems are technically classified as NP-hard, and we often resort to human-like heuristics to get their sub-optimal solutions. In case we really want to find an optimal path to a fixed goal of a problem instance in an enormous search space, the conventional A^* algorithm framework may be useful. The success of A^* algorithms depends on how to generate a maximally informed admissible version of *h-val*, the estimated distance to the goal state, such that it is not larger than but still as close as the unknown real distance to the goal. Recently we have suggested a method of generating a heuristic value with that property. To operate A^* algorithms in binary search fashions, some depth of fixed step backward states are pre-stored in disk, and the hashing schemes to handle efficiently pre-stored states must be designed to confirm fast the non-existence of a given state, not its existence, because the optimal path is there as soon as the existence of a state is confirmed. In this paper, state-nonexistence fast-confirming hashing schemes have been experimentally compared. The same pre-stored static backward states are also used for solving partial problems for the purpose of generating maximally informed admissible heuristic which guides the priority queue for A* algorithm in deciding which state to expand next. To show the validity of our method, it has been massively experimented for instances of Rubik's cube problem whose search space of states reachable from any given start state is known to cover $43*10^{18}$ states. The partial problems are experimentally compared, by varying forward search depths and tie-breaking functions, to show their effectiveness and efficiency in generating heuristic values.

Keywords: bidirectional A^*, state-nonexistence fast-confirming hashing, partial problem-based heuristic, dynamic forward search, static backward search

I. FINDING AN OPTIMAL PATH TO A FIXED GOAL OF A COMPLEX PROBLEM[1]

Most games and problems we face may be technically classified as NP-hard, which practically means that an optimal path to a goal state of a given problem becomes almost impossible to obtain as the size of a given problem instance becomes bigger, despite recent rapid hardware technology advances. Therefore, we normally seek their sub-optimal heuristic-based solutions. However, if we still need their optimal paths, the framework of A^*algorithm [1] [2] may be tried which theoretically produce optimal paths given sufficient time. The bidirectional A^* algorithm in [3] is shown in Fig. 1 to be used as the starting point of our discussion. This version is assumed to utilize for its backward search the pre-stored static state space including a fixed goal state.

```
unsignedintState::f() { return g_val+ h_val; }
boolBidirectional_A* {
priority_queue<State> OPEN;
set<State> CLOSED; // CLOSED is a set of states
START.g_val = 0; START.h_val = heuristic(START);
OPEN.push(START); // push START into OPEN
while ( OPEN is not empty ) {
  State P = OPEN.top();OPEN.pop();// state with min f
  if (P is in the pre-stored static backward search space) {
    // GOAL check included here
print the path from START through P to GOAL;
   return true; }
for (each child C of P) {
C.g_val = P.g_val + 1; C.h_val = heuristic(C);
if (C already exists as oldC in OPEN) {
if (C.f()>oldC.f())
      { OPEN.delete(oldC); OPEN.push(C);}
 } else if (C already exists as oldC inCLOSED)
      { CLOSED.delete(oldC); OPEN.push(C); }
   elseOPEN.push(C);
  } // end of for each child …
  CLOSED.add(P);
```

[1]This work is supported in part by Hongik University Research Fund 2013.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

```
  } // end of while ( true )
  return false; // no solution exists
} // end of bidirectional_A*
```

Fig.1General framework of bidirectional-A* algorithm

OPEN holding statesready to expand is a priority queue which returns the state with minimum *f_val*,which is the summation of *g_val* and *h_val*. *g_val* is the number of the steps from the initial state(START) to the current state, and*h_val* is the number of underestimatedsteps from it to the final state(i.e., GOAL).What matters most is how to generatean admissible (or nearly admissible) heuristic for calculating *h_val* such that its value is as large as possible but still not larger than real remaining steps.Korf tried a static pattern database [4], but we suggested a more complicated partial problem-based method, the outline of which has been described in [3]. This paper may be considered as its companion paper in which efficiency issues are experimentally treated regarding hashing methods and partial problem generation schemes.

II. SAMPLE PROBLEM SPACE FOR EXPERIMENTS

Our method of obtaining an optimal path to a given random start instance is general enough to be applied to any complex problem with a fixed goal state. However, just to clarify itsprocedure, we decided to utilize a well-known game problem, Rubik's cube, widely considered to be the world's best-selling toy[5]. It was estimated that 350 million cubes had been sold worldwide as of Jan. 2009 [6][7]. Humans can solve it in well under 100 moves with some methods [8][9], which are far from optimal and out of our concerns.

A. *God's Number*

A lower bound of 18 had been established by analyzing the number of effectively distinct move sequences of 17 or fewer moves, and finding that there were fewer such sequences than cube positions.In 1995 Michael Reid raised it to 20. The first upper bound was probably around 80 or so from the algorithm in one of the early solution booklets. In 1982, David Singmaster and Alexander Frey hypothesized that the number of moves needed to solve the Rubik's cube, given an ideal algorithm, might be in "the low twenties"[10].Computer search methods were used to demonstrate that any Rubik's cube can be solved in 26 moves[11], and in 22 moves[12], and in July 2010, researchers including Rokicki, with about 35 CPU-years of idle computer time donated by Google, proved the so-called "God's number" to be 20[13]. More generally, it has been shown that an n * n * n Rubik's cube can be solved optimally in the order of $n^2 / \log(n)$ moves[14].

Table I summarizes a history of God's number until it was shown to be 20[15].

TABLE I
GOD'S NUMBER IS 20[15]

Year	Lower bound	Upper bound	Notes and Links
	18	around 80	mathematical analysis, early solution booklet
1981	-	52	by Thistlethwaite
1982		might be low 20's	by Frey and Singmaster[10]
1990	-	42	by Kloosterman
1992	-	39	by Reid
1992	-	37	by Winter
1995	20	29	by Reid
2006	-	27	by Radu
2007	-	26	by Kunkle and Cooperman [11]
2008	-	22	by Rokicki and Welborn[12]
2010	-	20	byRokicki, and et. al. [13]

In this paper, we tested the effectiveness of our suggested method in solving an optimal or near optimal solution of a given Rubik's cube of some difficulty in 20 or less steps.

B. *The Problem Space and Its Backward Static Search Space*

Every state of the Rubik's cube can be defined by 48 tiles as in Fig. 2, excluding center tiles fixed during any move[16], though it has 6 faces each of which has 9 tiles. The goal state is the one with each face holding tiles of one color.The 48 tiles can be thought to be divided into 8 corners of 3 tiles and 12 edges of 2 tiles. Corners(Edges) move only to corner(edge) positions. A corner(edge) in a given position can be oriented in any of three(two) ways.The total number of states reachable from a given random state can be analytically calculated to be 43,252,003,274,489,856,000[4].

face number	face name
0	*T*
1	*K*
2	*L*
3	*F*
4	*R*
5	*B*

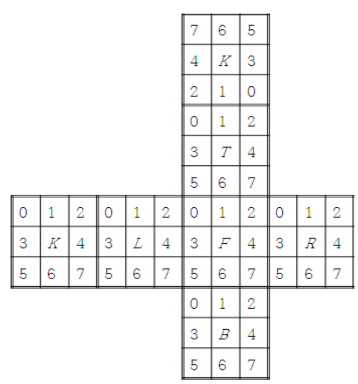

Fig. 2Face and tile number notations

If we pre-store *d_back*-step backward states, the forward search can be limited to the depth of 20-*d_back*, considering the total depth is limited to 20. Considering the state size of

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

40 bytes, Table II[15] shows that the pre-stored disk space is 0.3 Gigabytes for depth 6, 4.4 Gigabytes for depth 7, and 57.7 Gigabytes for depth 8. In our experiments, *d_back* was set to 7.

TABLE II
STATIC BACKWARD SEARCH SPACE AND DYNAMIC FORWARD SEARCH SPACE[15]

pre-stored backward search		**Forward depth**	**states in forward search space**
depth	**states**		
0	1	20	43,252,003,274,489,856,000
1	19	19	~43e18
2	262	18	~42e18
3	3,502	17	~13e18
4	46,741	16	~1.2e18
5	621,649	15	98,929,809,184,629,089
6	8,240,087	14	7,564,662,997,504,768
7	109,043,123	13	575,342,418,679,410
8	1,441,386,411	12	43,689,000,394,782
9	19,037,866,206	11	3,314,574,738,534
10	251,285,929,522	10	251,285,929,522

III. STATE-NONEXISTENCE FAST-CONFIRMING HASHING FOR PRESTORED STATIC SEARCH SPACE

The pre-stored static backward search space is so big that it is logical to store all the entries in hard disk space. It must be noted that as soon as we confirm that the matching state is in the pre-stored space, we have only to follow the path from it to GOAL, and we are done. Therefore what matters is not how fast we find a given state, but how fast we confirm that a given state does not exist.The experiments have been conducted on a specific domain here, but the same procedure can be applied more generally.

TABLE III
COLLISIONS FOR ADIFFERENT NUMBER OF BUCKETS

524,288 bucket HT		**100M bucket HT**		**500M bucket HT**	
bucket size	**buckets**	**bucket size**	**buckets**	**bucket size**	**buckets**
0	285,080	0	33,613,590	0	402035741
2	6	1	36,640,358	1	87663071
3	5	2	19,976,849	2	9565144
4	72	3	7,261,843	3	696162
5	16	4	1,982,473	4	38187
....	...	5	431,992	5	1641
410	256	6	78,773	6	53
411	259	7	12,218	7	1
412	268	8	1,644	8	0
413	266	9	233	9	0
...	...	10	25	10	0
5520	1	11	1	11	0
5560	1	12	1	12	0
total	524,288	**total**	100e6	**total**	500e6
empty rate	54.4%	**empty rate**	33.6%	**empty rate**	80.4%
Non-empty bucket size	455.85 ± 412.11	**Non-empty bucket size**	1.64 ± 0.86	**Non-empty bucket size**	1.11 ± 0.34

A. *Real World Collision Results for Different Numbers of Buckets*

For experimental purpose we stored $1.09*10^8$ states of 7-step space of Rubik's cube using hash tables by using problem-dependent realworld hashing functions, varying the number of buckets, and the experimental results are summarized in Table III.

For 0.5mega buckets, more than a half of buckets get empty, and no bucket turns out to contain a single state. For 100mega buckets, more than 70% of buckets are empty or contain just one entry position. If we utilize 500mega buckets, 80.4% of buckets get empty, and 17.5% of buckets contain just one entry position.

B. *Hash Table Structures*

For a hash table to be used for pre-stored backward searches it must be designed to confirm fast the nonexistence of a given state, which is quite different from the normal hashing designed to confirm fast where a given state is. Basically two step hashing is used, i.e., an array of *hash1* and an array of *hash2*, to be stored in disk. All the indexes to the pre-stored states are separately stored as an unsigned array (i.e. an array of *hash2*). *Hash1* is responsible for a bucket of states with the same hash function, and contains within it the bucket size (the number of state indexes) and the *start* position in the *hash2* array, which is 1st method. In method 2, for the bucket size equal to 1, the index itself is stored for the *start* position. In method 3, 1st index is stored in *hash1* itself, and the rest indices are in *hash2* array, except for the case of the bucket size 2, where 2nd index is stored for the *start* position.In method 4, 1st and 2nd indices are stored in *hash1* and the rest are in *hash2* array, except for the case of the bucket size 3, where 3rd index is stored for the *start* position.Method 3 and 4 require bigger '*struct*' sizes for *hash1* but reduces the number of indices to be stored in *hash2* array.

```
#typedef unsigned hash2
// hash2contains prestored backward state index
// array of hash2 is normally stored in disk.
//NBUCKETS is 524288 or 100M or 500M

<method 1>
structhash1 {
  unsignedbucketsize;
  unsignedstart; // pos of 1st hash2 entry for bucketsize>= 1
} hashtable[NBUCKETS];//normally in disk, not in memory

<method2>
structhash1 {
  unsignedbucketsize;
  union {
    unsignedstart; // 1st hash2 entry posfor bucketsize>= 2
    unsignedind;// 1st index for bucketsize == 1
  }
} hashtable[NBUCKETS];

<method 3>
structhash1 {
  unsignedbucketsize;
  unsignedind; // 1st index for bucketsize>= 1
  union {
    unsignedstart; //2nd hash2 entry posfor bucketsize>= 3
    unsignedind2; // 2nd index for bucketsize == 2 only
  }
} hashtable[NBUCKETS];

<method 4>
structhash1 {
  unsignedbucketsize;
  unsignedind; // 1st index for bucketsize>= 1
  unsignedind2; // 2nd index for bucket size >= 2
  union {
    unsignedstart; // 3rd hash2 entry posfor bucketsize>=4
    unsignedind3; // 3rd index for bucketsize == 3 only
  }
} hashtable[NBUCKETS];
```

C. *Analysis of Experimental Results*

All the data in this section is based on the experimental collision statistics in Table III. It must be kept in mind that around 4 giga bytes are already used for storing around 100 mega states of 7-step backward space.

1) *Memory Efficiencies:*Table IV summarizes how much more bytes are necessary to access the data based on hashing. Considering the size of our pre-stored data (4,361,724,920 bytes), 0.5 mega bucket table needs extra 0.440-0.444giga bytes(10.1~10.2%), 100 mega bucket table needs extra 1.090~1.633giga bytes(25.0~37.4%), and 500 mega bucket table needs extra 4.086-8.000giga bytes(93.7~183.4%).The memory overhead for hashing looks reasonable for 0.5 mega buckets or 100 mega buckets, but it gets very burdensome if we use 500 mega buckets. Table V shows the real sizes of buckets, i.e. *hash2* bucket sizes.

2) *Speed Efficiencies:*Table VI summarizes the average random accesses necessary to confirm the non-existence of a given state. To confirm that a given state is not stored, 2.12~2.75 accesses are needed for 100 mega bucket table, and 1.22~1.41 accesses are needed for 500 mega bucket table. For the latter, however, using more disk space does not much improve the average random accesses, and the method 3 with 1.22 accesses (or even method 2 with 1.24 accesses) looks a good choice. In case of 100 mega bucket table, the method 3 with 2.19 accesses looks reasonable. Table VII shows how long it will take to confirm the non-existence of one million given states, which is often the case with a complex problem like Rubik's cube. Table VIII summarizes the memory and speed of the 100 mega and 500 mega cases (method 3). Compared with 100 mega bucket table, 500 mega bucket table requires 4.72 giga byte space more, but can finish in 55.7% time.

TABLE IV
ADDITIONAL DISK SPACE IN BYTES FOR HASHING

Buckets	524,288	100e6	500e6
method 1	440,366,796	1,236,172,492	4,436,172,492
method 2	440,366,796	1,089,611,060	4,085,520,208
method 3	441,507,092	1,290,719,456	6,006,054,880
method 4	443,604,204	1,632,624,712	8,000,485,584

[Note] the above summarizes the additional storage for hashing pre-stored 4,361,724,920 byte data.

TABLE V
REAL SIZES OF *HASH2* BUCKETS

Buckets	524,288	100e6	500e6
empty rate	0.5443	0.3361	0.8041
rate of buckets whose size is 1	0	0.3664	0.1753
rate of buckets whose size is 2	1.14e-5	0.1998	0.1913
rate of buckets whose size is 3	0.95e-5	0.0726	0.0139
(method 1) real size of buckets whose size is >= 1	455.85± 412.11	1.64± 0.86	1.11± 0.34
(method 2) real size of buckets whose size is >= 2	455.85± 412.11	2.43± 0.71	2.08± 0.28
(method 3) real size of buckets whose size is >= 3	454.86± 412.11	2.32± 0.61	2.06± 0.24

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

(method 4) real size of buckets whose size is >= 4	453.87±412.11	2.25±0.54	2.04±0.21

TABLE VI
RANDOM ACCESSES TO CONFIRM THE NON-EXISTENCE OF ASTATE

Buckets	524,288	100e6	500e6
method 1	209.44±359.54	2.75±1.43	1.41±0.85
method 2	209.44±359.54	2.39±1.45	1.24±0.56
method 3	208.98±359.22	2.19±1.27	1.22±0.48
method 4	208.53±358.91	2.12±1.13	1.22±0.47

TABLE VII
ANALYTIC TIME(HR) TO PROCESS 1 MILLION RANDOM STATES(10MS DISK ACCESS)

Buckets	524,288	100e6	500e6
method 1	581.78	7.65	3.93
method 2	581.78	6.63	3.44
method 3	580.51	6.08	3.39
method 4	579.24	5.88	3.38

TABLE VIII
MEMORY AND TIME SUMMARY FOR METHOD 3

buckets	7-step pre-stored space	extra space for method 3 hashing	accesses to confirm state-nonexistence
100e6	4.36 giga bytes	1.29 giga bytes	2.19±1.27
500e6	4.36 giga bytes	6.01 giga bytes	1.22±0.48

IV. GENERATING PROPERLY INFORMED ADMISSIBLE HEURISTIC BASED ON PARTIAL PROBLEMS

We assume the problem has a fixed GOAL state and the static backward search space of states of some depth has been pre-computed, which is a one-time job. This may be classified as a method which generates and combines some partial solutions[3].

A. *Outline of a Partial Problem-based Method*

1) *Preliminary procedure*: First of all, the space *BSS* of states reachable (in the breadth first way) from GOAL must be built to be used for the backward search. The depth of the space, *d_back*, may be decided by considering the disk space reserved for storing static backward states.For example, we set *d_back* to 7 for Rubik's cube.This procedure may be summarized into following steps. (1) Generate the partial problems of the given problem instance, such that they are small enough to generate their optimal solutions fast, but big enough to generate large *h_val* usable for solving the original problem. Solve them in the framework of A^* for sufficient (say 30 or 50) random problems.(2) Select some partial problems whose max *h_val*is large enough. Let's call the static backward search space for*i*-th selected partial problem *BSS_PARTIAL*(i). The new heuristic is defined to be the maximum *h_val* of all selected partial problems.(3) Decide the proper forward depth, *d_for*, by considering the max *h_val* found for the partial problems subtracted by the pre-stored depth of static backward search space.
2) *Procedure for a Given Instance*: For each new problem instance, we have to construct *FSS_PARTIAL*(i), forward part of *SS_PARTIAL*(i). Note that we already have its backward part *BSS_PARTIAL*(i). Therefore the dynamically generated forward part should be much smaller than its backward counterpart, resulting in a limited *d_for* value. Consult [3] for the issues concerned with their generations. Given a problem instance, we construct some *FSS_PARTIAL*(i)'s, and we are ready to start the A^* algorithm.For each intermediate state we meet while running A^*algorithm, its *h_val* is set to the largest of all partial problem *h_vals*.

TABLE IX
SUMMARY OF PARTIAL CUBE SOLUTIONS FOR 25 SAMPLES WHOSE EFFECTIVE MOVES ARE 23.5± 3.0

	path len.	forward states	backward states	total states	time (sec)
corner cubes	9.0±0.9	47,149±9,713	14,242±14,220	61,391±20,762	0.4±0.1
edge cubes	12.0±0.9	603,075±249,028	260,923±221,965	814,716±425,869	5.2±3.0
0-1	11.4±1.1	297,720±204,996	63,098±57,056	360,818±244,362	1.3±0.8
0-1 +2/0	11.4±1.1	304,841±199,852	64,845±55,845	369,686±237,394	1.3±0.9
0-1 +2/4	12.0±1.4	506,661±229,703	202,372±180,062	709,033±375,408	2.5±1.5
0-1 +2/7	12.0±1.2	493,412±292,345	167,649±165,502	661,061±430,146	2.4±1.8
0-1 +2/67	12.4±1.2	1,168,857±997,672	472,184±203,580	1,641,041±1,119,939	7.7±7.3
0-1 +2/46	12.6±1.4	882,550±601,737	439,386±237,652	1,321,937±786,312	5.6±4.0
0-1 +2/25	12.5±0.9	547,993±249,162	266,310±220,572	814,304±433,557	3.0±1.7
0-1 +2/13	12.3±1.1	638,348±266,783	285,913±218,146	924,261±462,248	3.5±2.0
0-1 +2/257	12.6±1.2	1,404,869±1,307,434	477,427±214,144	1,882,295±1,428,153	9.8±11.3

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

0-1 +2/467	13.4 ±1.1	3,284,856± 2,770,177	625,519±27 2,393	3,910,375± 2,981,393	30.4± 33.6
0-1 + 2/0467	13.4 ±1.1	3,284,856± 2,770,177	625,519±27 2,393	3,910,375± 2,981,393	31.1± 34.7
0-1-2	14.8 ±1.3	12,418,911 ±5,838,039	8,640,670± 4,952,853	21,059,581 ±10,539,320	898.± 903.1

B. *Selection of Partial Problems*

Table IX summarizes partial problem efficiencies in terms of their path lengths, the number of states generated, and the total time. For these intermediate experiments, we utilized some heuristic the maximum value of which is 8. For instance of the partial cube notation, 0-1+2/67 denotes the faces 0 and 1 and two tiles (numbered 6 and 7) of the face 2 are used as a partial problem. 0-1+2/46 produces a very good result among two faces and two tiles partial problems, because tiles 4 and 6 of face 2 are the ones farther away from the faces 0 and 1. The more tiles we consider, the better path lengths(which is the partial problem *h-val*) we obtain at the cost of speed. We could utilize some 2 face plus 2 tile partial problems, but we decided to use 2 face ones, which would require more partial ones. Please note that these experiments are done for 25 random sample data whose effective moves are 23.5.

Table X summarizes the results obtained by combining partial problems of two faces. We used 50 random sample data with 50 effective moves. It should be noted that the data used for Table X are fully random and different from the ones for Table IX and the result comparisons must be done within the entries in the same table. The last case of 3 pairs is a good choice, which happens to use 3 partial problems of faces (a) 0 and 1, (b) 2 and 3, and (c) 3 and 5, implying that using 5 faces with one face overlapped is better than 6 faces non-overlapped.

TABLE X
*H_VAL*CALCULATED FOR PARTIAL PROBLEMS OF 50 RANDOM PROBLEM INSTANCES

h		**max**	**avg.**
a.	0-1	13	10.80
b.	2-3	12	10.98
c.	4-5	12	10.74
d.	1-2	13	10.84
e.	3-5	12	10.84
2-pairs	ab	13	11.44
	ac	13	11.22
	ad	13	11.14
	ae	13	11.38
3-pairs	abc	13	11.56
	abd	13	11.54
	abe	13	11.64
4-pairs	abcd	13	11.60
	abce	13	11.68

C. *Experimental Results*

For the experimental purpose, a set of Rubik's cube problem instances was generated and consistently used whose optimal path lengths are 10 to 14 steps. The number of states stored before solving the problem instance was counted.

Table XI summarizes the experimental results, the part of which was reported before [3], but we tried different tie-breaking rules. Basically for breaking ties, first additional heuristics (called *heu6* and *heu8* here) were tried and then last-come-first-out methods and first-come-first-out methods were applied. Generally speaking, stack-type tie-breaking rules worked better especially for harder problems. We won't delve into the additional heuristics here. Other experiments are all based on our suggested method with different max depths(*d_for*) of dynamic forward search space, set to 5-7. The currently used static space depth(i.e.*d_back*) 7 requires just 4 giga byte disk space, but it may be raised up to 10 to require 10 tera byte disk, acceptable in modern computers.The value of the dynamic search space depth, *d_for*, can be effectively raised as long as the memory capacity allows.

TABLE XI
EXPERIMENTAL RESULTS WITH 7-STEP PRE-STORED BACKWARD STATES

(a) forw. heuristic=heu6

#steps	**10**	**11**	**12**	**13**	**14**
forward states stored	957	33,987	628,964	9,558,799	> 50e6

(b) forw. heuristic=myh(d_for=5)

#steps		**10**	**11**	**12**	**13**	**14**
states stored for each two-face partial prob. (46741+α)		363	296	174	137	142
forward states stored :queue -type tie breaker	**5-bfs inside**	63	237	837	1,321,397	> 50e6
	5-bfs /heu6 inside	63	237	837	7,032,237	> 50e6
	5-bfs /heu8 inside	63	225	837	4,570,108	> 50e6
forward states stored :stack -type tie breaker	**5-bfs inside**	63	228	894	1,450,964	> 50e6
	5-bfs /heu6 inside	63	228	894	3,162,174	>

						50e6
	5-bfs /heu8 inside	63	228	894	3,186,957	> 50e6

(c) forw. heuristic=myh(d_for=6)

#steps		**10**	**11**	**12**	**13**	**14**
states stored for each two-face partial prob. (621649+α)		3980	3361	2113	1839	1904
forward states stored :queue -type tie breaker	**6-bfs inside**	120	504	1011	8896	46,545,179
	6-bfs /heu6 inside	120	504	1011	8896	> 50M
	6-bfs /heu8 inside	120	492	1011	8896	> 50M
forward states stored :stack -type tie breaker	**6-bfs inside**	93	420	1665	10186	11,615,151
	6-bfs /heu6 inside	93	420	1665	10186	47,422,873
	6-bfs /heu8 inside	93	477	1665	10186	4,025,489

(d) forw. heuristic=myh(d_for=7)

#steps		**10**	**11**	**12**	**13**	**14**
states stored for each two-face partial prob. (8,240,087+α)		42718	38375	26516	23549	25143
forward states stored :queue -type tie breaker	**7-bfs inside**	510	10338	12747	96696	198515
	7-bfs /heu6 inside	510	10338	12747	96689	198245
	7-bfs /heu8 inside	510	8406	12720	97832	198383
forward states stored :stack -type tie breaker	**7-bfs inside**	657	10938	5064	174898	89261
	7-bfs /heu6 inside	657	10950	5064	174353	89261
	7-bfs /heu8 inside	657	10968	5064	175002	127414

V. CONCLUSION

Many problems may be stated in the framework of binary search A^* algorithm with the pre-stored backward search space.First of all, the design of hashing schemes,which fast confirms the non-existence of a state, not its existence, is necessary to effectively utilize the pre-stored space. In addition, to practically utilize A^*whichguarantees the final path optimality we have to devise informed admissible heuristic for a given specific problem. Accordingly, how to generatepartial problems which may suggest that kind of heuristicmay be the key to the success of A^* given a problem instance.

Our bidirectional search paradigm was massively tested for the practical domain of Rubik's cube. The hashing schemes for fast confirming the non-existence of a state in the pre-stored backward space were experimentally compared. The generation of partial problems was also tested by varying the dynamic search depth for different tie-breaking methods.

Though a specific domain was used for experiments, the same procedure can to applied to a broader spectrum of complex problems with a fixed goal in finding their optimal (or almost optimal) paths efficiently.

ACKNOWLEDGMENT

This work is supported in part by Hongik University Research Fund 2013.

REFERENCES

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. on Science and Cybernetics (2), pp.100-107, 1968.

[2] G. Luger, "Heuristic Search," Ch. 4, *Artificial Intelligence*, 6ed., Pearson, 2008.

[3] K. Lee and H. Kim, "Analyzing and Combining Partial Problem Solutions for Properly Informed Heuristic Generations," IISTE Computer Engineering and Intelligent Systems 3(11), pp1-8, 2012.

[4] R.Korf, "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases," AAAI/IAAI, pp.700-705, 1997.

[5] "Rubik's Cube 25 years on: crazy toys, crazy times," The Independent(London), Aug. 16, 2007.

[6] W. L. Adams, "The Rubik's Cube: A Puzzling Success," TIME, Jan, 2009.

[7] A. Jamieson, "Rubik's Cube inventor is back with Rubik's 360," The Daily Telegraph (London), Jan. 2009.

[8] P. Marshall, "The Ultimate solution to Rubik's cube," http://helm.lu/cube/MarshallPhilipp/, 2005.

[9] D.Singmaster, "Notes on Rubik's Magic Cube," Harmondsworth, Eng: Penguin Books, 1981.

[10] A. Frey and D.Singmaster, Handbook of Cubik Math, Enslow Publishers, 1982.

[11] D. Kunkle and C. Cooperman, "Twenty-Six Moves Suffice for Rubik's Cube," Proc. of the International Symposium on Symbolic and Algebraic Computation(ISSAC '07), ACM Press, 2007.

[12] T.Rokicki, "Twenty-Two Moves Suffice," http://cubezzz.dyndns.org/drupal/?q=node/view/121, Drupol, Aug.12 2008.

[13] J. F. Flatley, "Rubik's cube solved in twenty moves, 35 years of CPU time," Engadget, Aug. 9, 2010.

[14] E.Dermaine, M. L. Dermaine, S.Eisenstat, A.Lubiw, and A. Winslow, "Algorithms for Solving Rubik's Cubes," arTiv:1106.5736v, 2011.

[15] God's Number is 20, http://www.cube20.org/, 2012.

[16] M. W. Dempsey, "Growing up with science: The illustrated encyclopedia of invention," London: Marshall Cavendish, pp.1245, 1988.

ISSN (Print) : 2319-5940
ISSN (Online) : 2278-1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 5, May 2013

BIOGRAPHY

Kee-cheol Lee He was born in Seoul, Korea in 1955. He received a BS degree in electronic engineering from Seoul National University in 1977, a MS degree in computer science from Korea Advanced Institute of Science in 1979, and a Ph.D in electrical and computer engineering from University of Wisconsin-Madison in 1987. Since 1989, he has been on the faculty of computer engineering department, Hongik University, Seoul, Korea, and currently he is a professor. His academic and research interests cover the fields of artificial intelligence, machine learning, and information retrieval.