Analysing and Combining Partial Problem Solutions for Properly Informed Heuristic Generations

Kee-cheol Lee1* and Han-gyoo Kim
Computer Engineering Department, Hongik University, Seoul, Korea 121-791
* E-mail of the corresponding author: kclee1@hongik.ac.kr

Abstract
Finding an optimal path to the fixed goal state of a problem instance lying in an enormous search space may be described in the framework of the conventional A* algorithm. However, the estimated distance to the goal state, so called $h_value$, must be generated by an admissible heuristic such that it is not larger than but still as close as the unknown real distance to the goal. In this paper, we suggest a method of generating a heuristic with that property. After analyzing a number of devised partial problems, some are selected to be combined to produce a properly informed heuristic. In solving a complex problem with a fixed goal, some depth of fixed backward states is pre-stored. Those static backward states are also used for partial problem backward searches. For a given problem instance, the forward search is first performed for each of its partial problem. The dynamically generated space is combined with the static search space to produce the temporary search space, which is used to aid in the generation of each state heuristic for the course of problem solving. A novel method of constructing the temporary search space for each partial problem is suggested, in which each forward state found in the static backward space is back-propagated and propagated in the forward space. To show the effectiveness of our method, it has been massively experimented for instances of Rubik’s cube problem of some difficulty whose search space of states reachable from any given start state is known to cover $43 \times 10^{18}$ states, the number of which even an 64-bit unsigned integer cannot hold.

Keywords: A*, admissible heuristic, partial problems, dynamic forward search, static backward search, Rubik’s cube

1. General Framework for Solving a Complex Problem
Most complex problems are technically classified as NP-hard problems, which mean that as the size of a problem instance gets larger, its optimal solution gets virtually impossible to obtain. That is why heuristic-based knowledge is generally used for generating a good solution instead of the optimal one. However, to get the optimal (or even near optimal) one for a somewhat larger size of a problem instance, the framework of A* algorithm may be tried. Figure 1 describes the A* algorithm (Nilsson et al. 1968; Luger 2008) modified to utilize the pre-stored static backward space of states. Two auxiliary functions are additionally used in the algorithm.

```c
void State::set_h() { /* heuristic for calculating h_value */ }
unsigned int State::get_f() { return g_value + h_value; }
bool Modified_A* {
    priority_queue<State> OPEN;
    set<State> CLOSED; // CLOSED is a set of states
    START.g_value = 0; START.set_h(); OPEN.push(START); // START into OPEN
    while ( true ) {
        if (OPEN.isempty()) return false; // no solution exists
        State P = OPEN.top(); OPEN.pop(); //get state with max f
        if (P in pre-stored static backward search space) {
            // back-propagation and propagation
        }
    }
}
```

1 This work is supported in part by Hongik University Research Fund 2012.
print path from START through P to GOAL; return true; }
for (each child C of P) {
    C.g_value = P.g_value + 1; C.set_h();
    if (C exists as oldC in OPEN or CLOSED) {
        if (C.get_f() > oldC.get_f()) {
            delete(oldC); OPEN.push(C); }
    } else OPEN.push(C);
} // end of for each child …
CLOSED.delete(P);
} // end of while (true)
} // end of Modified_A*

Figure 1. General Framework of the Modified A* algorithm

Let START denote the state of the given instance and GOAL be the static final state. In this version, GOAL
is one of the states in the pre-stored static backward space, and does not explicitly appear in Figure 1. Each
state has g_value, the number of steps from START to it, and h_value, the number of estimated steps from
it to GOAL. (Those two values may be differently defined in terms of moving costs instead of the number
of steps. However, without loss of generality we assume the latter.) The sum of the two values, f_value
(returned by the function get_f()) is used to guide OPEN, the max priority queue holding the states ready to
expand. If the h_value of a state could always be set to its unknown actual number of steps (called
h*_value), A* algorithm would reduce to an ideal analytic solution. The main issue here is how to derive the
admissible (or nearly admissible) heuristic, i.e. function set_h(), such that its value is as large as possible
but still not larger than h*_value. A static pattern database (Korf 1997) may help, but we suggest a more
complicated method.

2. Generating Properly Informed Admissible Heuristics

We assume the problem has the fixed GOAL state and the static backward search space of states of some
depth has been pre-computed, which is just a one-time job. This may be classified as a method which
generates and combines some partial problem solutions (Lee & Kim 2012).

2.1 Preliminary procedure for the problem domain

For the given problem domain (e.g., Rubik’s cube), the following preliminary procedure must be done to
decide which partial problems must be used, and to prepare the corresponding static spaces.

2.1.1 Construction of static backward search space

Make a space BSS of states reachable (in the breadth first way) from GOAL to be used for backward
searches. Its depth, d_back, may be a design parameter and can be decided considering the disk space
reserved for storing static backward states.

2.1.2 Design of partial problems and their indexing schemes

Generate partial problems of the given problem. Solve them in the framework of A* for some (say 50)
random problems. The partial problems must be small enough to generate their optimal solutions fast, but
big enough to generate large h_values usable for solving the original problem. The different hashing
scheme suitable for each partial problem should be designed to access the same pre-stored static search
space. Its details will not be discussed in this paper.

2.1.3 Selection of partial problems and construction of BSS_PARTIAL(i), the static backward part of
SS_PARTIAL(i)
Select some partial problems whose max $h_values$ are large enough. Let’s call the backward search space for the $i$-th selected partial problem $BSS_{PARTIAL}(i)$. The new heuristic is defined to be the maximum of the $h_values$ of all selected partial problems.

2.1.4 Design of the proper depth of the dynamic forward search space

Decide the proper forward depth, $d_{for}$, by considering the max $h_value$ found from partial problems subtracted by the pre-stored depth of static backward search space.

2.2 Procedure for a given instance

For each new problem instance, the following procedure can be used.

2.2.1 Construction of $FSS_{PARTIAL}(i)$, forward part of $SS_{PARTIAL}(i)$

We already have $BSS_{PARTIAL}(i)$, the static part of $SS_{PARTIAL}(i)$. Now the dynamic forward part of $FSS_{PARTIAL}(i)$ is built in this stage. While building $FSS_{PARTIAL}(i)$, any newly generated state for the $i$-th partial problem is tested if it already exists in $BSS_{PARTIAL}(i)$. If that is the case, the known $h_value$ of the old one is the new $h_value$ of the new one, and it is back-propagated back to START in the forward search space. In addition, after $FSS_{PARTIAL}(i)$ is temporarily constructed, all $h_values$ of the states from START to the final depth of the forward space must be propagated such that the $h_values$ of any adjacent states cannot be different by more than 1. The algorithm which reflects this is as follows.

```cpp
for (int i = 0; i < #selected_partial_problems; i++) {
    // make a dynamic forward search space $FSS_{PARTIAL}(i)$
    START.h_value = d_for + d_back; START.g_value = 0; // h set to max value; g set to #steps from START
    vector<State> V; V.addrear(START); // V is used as a big array; start form START
    ind = 0; State Cur, C, P;
    while ( ind < V.size() ) {
        CUR = V[ind++];
        if (CUR.g_value < d_for) { // init & push each child except for deepest depth
            for (each child C of CUR) {
                CUR.h_value = CUR.h_value-1; C.g_value = CUR.g_value+1; V.addrear(C);
            }
        } else if (CUR already exists in $BSS_{PARTIAL}(i)$) {
            CUR.h_value = the depth of the stored state;
            C = CUR;
            // back-propagate the newly acquired $h_value$
            while (C != START) {
                Let P be the parent state of C;
                if (P.h_value <= (C.h_value + 1)) break; // no more back-propagation
                P.h_value = C.h_value+1; // lower P’s $h_value$
                C = P; // for one step back
            }
        }
    }
    // Now propagation begins
    for (int ind = 1; ind < V.size(); ind++) {
        C = V[ind]; Let P be the parent state of C;
        if (C.h_value > (P.h_value+1)) C.h_value = P.h_value + 1; // lower C’s $h_value$
    }
}
```

3
2.2.2 \textit{h\_value} calculating function, \textit{set\_h}

Please note that the above procedure is done once for a given problem instance. Those spaces for partial problem instances are used throughout the problem solving. The heuristic-calculating function utilizes those fixed spaces. For each state of a problem space, the largest of all partial problem \textit{h\_values} is the wanted \textit{h\_value}. That value and \textit{g\_value}, the number of steps (i.e., depths) from START to the current state, are added to produce the \textit{f\_value} of a state. The OPEN priority queue, which contains the states ready to expand, is designed such that its \textit{top} function returns the state with the largest \textit{f\_value}.

```cpp
void State::set_h()
{
    h_value = 0;
    for (int i = 0; i < #selected_partial problems; i++)
    {
        if (this state in SS\_PARTIAL(i)) new_h = its stored h\_value;
        else new_h = d\_back + 1; // defaulted to 8 for d\_back set to 7
        if (new_h > h_value) h_value = new_h;
    }
}
```

![Figure 2. TFace and Tile Number Notation](image)

3. The problem Space for Experiments

Just to clarify the validity of our method, we utilized a well-known game problem, Rubik’s cube, widely considered to be the world’s best-selling toy (Indep. 2007). As of January 2009, 350 million cubes had been sold worldwide (Adam 2009)(Jamieson 2009). A number of solutions have been developed which can solve the cube in under 100 moves (Marshall 2005; Singmaster 1981), which are far from optimal, and we are not interested in them.

Frey & Singmaster (1982) that the number of moves needed to solve any Rubik’s cube, given an ideal algorithm, might be in “the low twenties”. Kunkle & Cooperman (2007) used computer search methods to demonstrate that any Rubik’s cube can be solved in 26 moves or fewer. Rokicki (2008) lowered that number to 22 moves, 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 (Flatley 2010). More
generally, it has been shown that an n x n x n Rubik’s cube can be solved optimally in the order of \( n^2 \log(n) \) moves (Dermaine et al. 2011).

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.

3.1 The problem space

The Rubik’s cube has 6 faces each of which has 9 tiles. Because center tiles are fixed during any move, every cube state can be defined by 48 tiles as in Fig. 2. For the puzzle to be solved, each face must be returned to consisting of one color (Dempsey 1988).

The 48 tiles can be divided into two groups, 8 corner cubies of three tiles and 12 edge cubies of two tiles. Corner (Edge) cubies move only to corner (edge) cubie positions. The entire space is also known to consist of 12 separate but isomorphic sub-graphs with no legal moves between them (Korf 1997). Therefore, the total number of states reachable from a given state is \( (12!*2^{12})*(8!*3^8)/12 = 43,252,003,274,489,856,000 \), which is greater than the max number even an unsigned 64-bit integer can hold (i.e., \( 17.592*10^{18} \)).

Because we know the number of reachable states and God’s number is 20, the average branching factor \( b \) from depth 0 to 20 can be calculated to be 9.536 by solving the equation of the sum of 21 terms of the geometric sequence, i.e., \( 1 + (b^{11} - 1)/(b - 1) = 43.252 * 10^{18} \).

3.2 Storing the static space for backward search

The bi-directional search helps to reduce the search space. For example, Table 1 (Rokicki 2010) shows that if 10-step forward search space and 10-step backward search space is used, we may deal with the space of \( 0.5e12 \) states, whereas 20-step forward search would require a space of \( 43*e18 \) states. That reduction ratio corresponds to 1 versus one hundred million. Each row in the table shows the number of states generated for problem solving. For example, backward space of depth 9 can be used with forward space of 11.

Because the GOAL state is fixed, we can pre-calculate the space of backward search and store it in disk (or

<table>
<thead>
<tr>
<th>depth</th>
<th>#states</th>
<th>depth</th>
<th>#states</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>20</td>
<td>43,252,003,274,489,856,000</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>19</td>
<td>~43e18</td>
</tr>
<tr>
<td>2</td>
<td>262</td>
<td>18</td>
<td>~42e18</td>
</tr>
<tr>
<td>3</td>
<td>3,502</td>
<td>17</td>
<td>~13e18</td>
</tr>
<tr>
<td>4</td>
<td>46,741</td>
<td>16</td>
<td>~1.2e18</td>
</tr>
<tr>
<td>5</td>
<td>621,649</td>
<td>15</td>
<td>98,929,809,184,629,089</td>
</tr>
<tr>
<td>6</td>
<td>8,240,087</td>
<td>14</td>
<td>7,564,662,997,504,768</td>
</tr>
<tr>
<td>7</td>
<td>109,043,123</td>
<td>13</td>
<td>575,342,418,679,410</td>
</tr>
<tr>
<td>8</td>
<td>1,441,386,411</td>
<td>12</td>
<td>43,689,000,394,782</td>
</tr>
<tr>
<td>9</td>
<td>19,037,866,206</td>
<td>11</td>
<td>3,314,574,738,534</td>
</tr>
<tr>
<td>10</td>
<td>251,285,929,522</td>
<td>10</td>
<td>251,285,929,522</td>
</tr>
</tbody>
</table>
physical memory, depending on its size). Contrarily, the forward space is different for each problem instance and must be dynamically generated. If we pre-store $d_{back}$-step backward states, the forward search can be limited to the depth of $20-d_{back}$, because we now know that the total depth can be safely limited to 20, God’s number. Considering one state needs 40 bytes, we can deduce from Table 1 that the pre-stored disk space is 0.3 Gigabytes for depth 6, 4.4 Gigabytes for depth 7, 57.7 Gigabytes for depth 8, 761.5 Gigabytes for depth 9, and 10.1 Terabytes for depth 10. This may be time-consuming, but it should be noted that it should be done and stored once for the Rubik’s cube domain. In our experiments, the value of $d_{back}$ is set to 7, and moderate 4.4 Gigabytes are used for that.

<table>
<thead>
<tr>
<th></th>
<th>a. 0&amp;1</th>
<th>b. 2&amp;3</th>
<th>c. 4&amp;5</th>
<th>d. 1&amp;2</th>
<th>e. 3&amp;5</th>
<th>2-pairs</th>
<th>3-pairs</th>
<th>4-pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>13</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>avg</td>
<td>10.80</td>
<td>10.98</td>
<td>10.74</td>
<td>10.84</td>
<td>10.84</td>
<td>11.44</td>
<td>11.22</td>
<td>11.14</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>1.00</td>
<td>0.91</td>
<td>0.84</td>
<td>0.78</td>
<td>0.78</td>
<td>0.83</td>
<td>0.73</td>
<td>0.85</td>
</tr>
</tbody>
</table>

4. Selection of Partial problems

Table 2 summarizes the results obtained by combining partial problems of two faces. Face numbers 0-5 denote Top, Back, Left, Front, Right, and Bottom faces. For the experimental purpose, fifty random cubes generated by sufficiently many moves have been utilized. All the partial problems of two faces have been solved in not more than 13 steps. Out of those 5 basic partial problems, some pairs of their $h$ values have been combined. The last case of 3 pairs is a good choice in that its average $h$ value is sufficiently high (11.64), with relatively low (0.62) standard deviation ($\sigma$), which indicates the stability of the method. That happens to be the case of using 3 partial problems of faces (a) 0 & 1, (b) 2 & 3, and (c) 3 & 5, implying that using 5 faces with one face overlapped may be better than using all of 6 faces.

5. Experimental Results

For the experimental purpose, a series of Rubik’s cube problem instances with their optimal lengths 8-14 were consistently used. We counted the number of states stored while running the main $A^*$ algorithm.

Table 3 summarizes the experimental results. The first experiment was done using a heuristic (called heu6 here) which is the number of misplaced tiles of all the edge cubies divided by 4, and whose max value is limited to 6. We won’t delve into its details 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 table shows that as the value of $d_{for}$ grows, harder problems may be solved with a much smaller number of states stored. The value of the static space depth, $d_{back}$, is currently set to 7 such that it requires just 4 Giga byte disk space to hold pre-stored states, but practically we could raise it to even 10 because it requires 10.1 Tera byte disk, which is acceptable in modern computers, and because its construction is just one time job. The value of the dynamic search space depth, $d_{for}$, can be effectively raised as long as memory capacity allows.
### Table 3. Experimental results with 7-step pre-stored backward states

<table>
<thead>
<tr>
<th>(a) forw. heuristic=heu6</th>
<th>(b) forw. heuristic=myh(d_for=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>forward states stored</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>957</td>
</tr>
<tr>
<td>11</td>
<td>33,987</td>
</tr>
<tr>
<td>12</td>
<td>628,964</td>
</tr>
<tr>
<td>13</td>
<td>9,558,799</td>
</tr>
<tr>
<td>14</td>
<td>&gt; 50M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c) forw. heuristic=myh(d_for=6)</th>
<th>(d) forw. heuristic=myh(d_for=7)</th>
</tr>
</thead>
<tbody>
<tr>
<td># steps</td>
<td>states stored for partial problems of faces</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
</tr>
<tr>
<td>8</td>
<td>5,363</td>
</tr>
<tr>
<td>9</td>
<td>3,914</td>
</tr>
<tr>
<td>10</td>
<td>5,069</td>
</tr>
<tr>
<td>11</td>
<td>3,897</td>
</tr>
<tr>
<td>12</td>
<td>2,805</td>
</tr>
<tr>
<td>13</td>
<td>2,240</td>
</tr>
<tr>
<td>14</td>
<td>2,407</td>
</tr>
</tbody>
</table>

### 6. Conclusion

The optimal solution of a complex problem is very hard to obtain as the size of the problem instance grows. Therefore its heuristic-based suboptimal solution is often tried. If its optimal or near optimal solution is really necessary, it can be tried in the framework of A* algorithms. The admissible heuristic for A*, $h_{\text{value}}$, is the guess of the actual number of steps, $h^*_{\text{value}}$, from the current state to the goal state and must not exceed it. The generation of admissible (or almost admissible) and sufficiently informed heuristic, $h_{\text{value}}$, for each state is the most important, given a problem instance.

We suggested a bidirectional search paradigm with the static backward search space and the dynamic forward search space which utilizes the heuristic value generated by combining each devised partial problem instance heuristic. For a problem domain (e.g., Rubik’s cube), some partial problems are selected and their hashing schemes are designed to maintain their own static backward spaces. For every problem instance given, its dynamic forward space of states is generated to be combined with its own static space. If any new state being generated in the forward space of a partial problem already exists in the corresponding backward space, its old $h_{\text{value}}$ is back-propagated to the start state, and propagated in the dynamic...
forward search space. The newly combined space for each partial problem is used to calculate its own $h_{value}$. The maximum of all the $h_{values}$ recommended by each partial problem is used as the $h_{value}$ of each newly generated state of the given problem instance.

To show the effectiveness of our suggested method, it has been successfully experimented for a series of Rubik’s cube problem instances of some difficulty.

References


Kee-cheol Lee He was born in Seoul, Korea on Feb. 21, 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 degree in electrical and computer engineering from University of Wisconsin-Madison in 1987. Since March 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.

Han-gyoo Kim Prof. H. Kim was born in Seoul, Korea in 1959. He received B.S. degree in mechanical engineering from Seoul National University in 1981, and his Ph. D in computer science from University of California at Berkeley in 1994. Since August 1994, he has been on the faculty of computer engineering department, Hongik University, Seoul, Korea. His areas of research include networked storage systems, scalable information retrieval systems, and high speed large scale big data systems.