A Perfect Lookup Table Evaluation Function for the Eight-Puzzle

Paul E. Utgoff
Sharad Saxena

COINS Technical Report 87-71

August 4, 1987

University of Massachusetts
Amherst, MA 01003

Abstract

The eight-puzzle is a classic problem that is used for testing, illustrating, and comparing problem solving algorithms. Brute force search is expensive. Several heuristics have been suggested for heuristic search. This report describes a perfect evaluation function for the eight-puzzle. The evaluation function is represented as a lookup table. Such an evaluation function can be helpful for study of the eight-puzzle or other small puzzles.

Contents

1. Introduction

The eight-puzzle consists of 8 square tiles lying in a 3x3 grid. The tiles are numbered 1 through 8. A legal move consists of sliding a tile that is horizontally or vertically adjacent to the empty location into the empty location. The standard goal state is the particular configuration shown in figure 1. An eight-puzzle problem is generated by scrambling the tiles using only legal moves. The objective is to solve the puzzle by identifying a sequence of moves that restores the tiles to the goal state configuration.

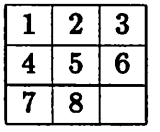

Figure 1: Goal State for the Eight-Puzzle

2. Perfect Evaluation Function

An evaluation function maps a problem state to an estimate of the remaining work to be done to reach a goal state. A *perfect* evaluation function is one in which the estimate is correct. Many imperfect evaluation functions have been devised for the eight-puzzle, such as the total number of misplaced tiles, and the sum of the Manhattan distance of each tile from its correct location in the goal state (Nilsson, 1971). This section describes a perfect evaluation function, represented as a lookup table.

There are 9! possible ways to lay 8 tiles into a 3x3 grid. However, given the rule for moving a tile, these 9! states are partitioned into two subsets. One subset contains all the states from which the goal state can be reached in 0 or more steps. The other set contains the states from which the goal state cannot be reached.

For each of the 9!/2 legal states, a perfect evaluation function maps the state to the number of moves needed to reach the goal state. It should be noted that with the existence of a perfect evaluation function, hill-climbing is a sufficient search algorithm. A perfect evaluation function was generated as a lookup table in an exhaustive manner using the procedure given in figure 2.

The table generation procedure is straight forward except for mapping a state to an index for the purpose of indexing the lookup table. The algorithm for mapping the permutations one to one onto a unique interval of integers is taken from (Reingold et al, 1977). The generation algorithm creates the lookup table in 124 cpu minutes on a VAX 750 under Berkeley 4.2bsd. The code is a combination of Franz Lisp and C.

3. Analysis

The lookup table allows some simple analysis. A distribution of the number of states for each value is shown in figure 3. Recall that the value is the number of steps needed to reach the goal state. For example, there are two distinct problem states that require

1. Initialize the lookup table with each entry as undefined.
2. Set *open* list to contain a single element that is the goal state G with value 0. An element is a (state,value) pair.
3. Remove the first (state,value) pair from *open*, call it (s, v).
4. Define $lookup(s)$ to have value v.
5. Generate the successors of s, call them *succ*, associating with each one the value $v + 1$.
6. For each $succ_i$, if $lookup(succ_i)$ is undefined, then append the element $(succ_i, v + 1)$ to the end of *open*.
7. If *open* is empty then stop, else go to 3.

Figure 2: Procedure for generating lookup table

exactly one step to be solved. Note that the worst case problem requires 31 moves. The average problem requires 21.9724 moves.

A further observation is that one can compute the size of a restricted problem space. For example, if one is limited to problems that require at most 10 steps, then the problem space contains only 706 states, the result of

$$\sum_{i=0}^{10} states(i)$$

with $states(i)$ returning the total number of states of length i, as indicated in the distribution.

References

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill.

Reingold, E. M., Nievergelt, J. and Deo, N. (1977). *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall.

Value	States
0	1
1	2
2	4
3	8
4	16
5	20
6	30
7	62
8	116
9	152
10	286
11	306
12	748
13	1024
14	1803
15	2512
16	4485
17	5638
18	9520
19	10878
20	16903
21	17110
22	23952
23	20224
24	24047
25	15578
26	14560
27	6274
28	3010
29	760
30	221
31	2

Figure 3: Distribution of states by solution length

A Program Code

A1 File table.c

```
#include <sys/file.h>

typedef unsigned char uchar;

static uchar *table = (uchar *) 0;
static long tablesize = 0;

inittable(tsize)
  int *tsize;
{ int i;

  if (*tsize<0)
    return(-1);
  else
    { if (table) free(table);
      tablesize = *tsize;
      table = (uchar *) malloc(tablesize);
      for (i=0; i<tablesize; i++) table[i]=0;
      return(tablesize);
    };
}

restoretable()
{ int fd;
```

```
  fd=open("eight.table",O_RDONLY,0660);
  if (fd)
    { if (table) free(table);
      read(fd,&tablesize,sizeof(long));
      table = (uchar *) malloc(tablesize);
      read(fd,table,tablesize);
      close(fd);
      return(tablesize);
    }
  else
    return(-1);
}

savetable()
{ int fd;

  fd=open("eight.table",O_CREAT|O_WRONLY,0660);
  if (fd)
    { write(fd,&tablesize,sizeof(long));
      write(fd,table,tablesize);
      close(fd);
      return(tablesize);
    }
  else
    return(-1);
}

puttable(index,val)
  int *index,*val;
{ if (table && 0<=(*index) && (*index)<tablesize && table[*index]==0)
    { table[*index]=(*val)+1;
      return(table[*index]-1);
    }
  else
    return(-1);
}

gettable(index)
  int *index;
{ if (table && 0<=(*index) && (*index)<tablesize)
    return(table[*index]-1);
  else
    return(-1);
}
```

A2 File table.l

```
(cfasl "table.o" '_inittable 'init-table "i")
```

```
(getaddress '_restoretable 'restore-table "i")

(getaddress '_savetable 'save-table "i")

(getaddress '_puttable 'put-table "i")

(getaddress '_gettable 'get-table "i")

(cfasl 'indexfns.o '_state_to_index 'state-to-index "i")

(getaddress '_index_to_state 'index-to-state "i")
```

A3 File indexfns.c

```
/* returns the lexicographic number of a permutation in perm of size length */
#include <stdio.h>
state_to_index(perm)
  long *perm;
{ int state[16],length;

        int index_value,element,index,factorial,temp,k;
  length=0;
  while (perm)
    { state[length++] = * ( (int *) (*((long *) perm+1)));
      perm = (long *) *perm;
    };
        index_value = 0;
        for (element = 0; element < length ; element ++)
        {
                index = state[element] - 1 ;
                factorial = length - element - 1 ;
                temp = factorial ;
                while (temp > 1) factorial *= (--temp) ;
                index_value += index * factorial ;
                for (k = element + 1 ; k < length ; k++)
                        if( state[k] > state[element])
                                state[k]--;
        }
        return(index_value);
}

index_to_state(index,length,perm)
  long *index,*length,*perm;
{
int state[16],factorial,number,remainder,pos,elements,i,j,k;
        elements = *length - 1;
        number = *index ;
```

```
        factorial = 1 ;
        pos = 0 ;
        while (*length > 1)
                {
                for (i = 2 ; i <= *length ; i++)
                        {
                        factorial *= i - 1 ;
                        remainder = number % i ;
                        number = number/i ;
                        }
                state[pos] = remainder;
                (*length)-- ;
                pos += 1;
                *index -= (remainder * factorial);
                number = *index;
                factorial = 1;
                }
        state[elements] = 1 ;
        for(i = elements - 1 ; i >= 0 ; i--)
        {
                for(j = i+1 ; j <= elements ; j++)
                if(state[j] > state[i]) state[j] += 1 ;
                state[i] += 1;
        }
 k=0;
 while (perm)
   { * ( (long *) (*((long *) perm+1))) = state[k++];
     perm = (long *) *perm;
   }
  return(k);
}
```

A4 File tablesum.c

```
#include <stdio.h>
main()
{ int i,n,count[254],maxcnt,j,total;

  total=0;
  maxcnt=0;
  for (i=0; i<254; i++) count[i]=0;

  n=restoretable();
  printf("Table size = %d\n",n);

  for (i=0; i<n; i++)
    { j=gettable(&i);
      if (j>=0) count[j]++;
```

```
      if (j>maxcnt) maxcnt=j;
    };

  for (i=0; i<=maxcnt; i++)
    { printf("count[%2d] = %6d\n",i,count[i]);
      total += count[i];
    };

  printf("Table holds %d elements\n",total);

  exit(1);
}
```

A5 File positions.l

```
(eval-when (compile) (includef '/user/lrn/utgoff/lisp/lisp.l))

(declare (macros t)
         (special ~goal
                  ~instances
                  ~top-row
                  ~bottom-row
                  ~left-column
                  ~right-column
                  ~row-size
                  ~board-size
                  ~table-size
                  ~max-elements
                  ~perm-hunk
                  ~debug
                  poport
                  TERM))

(setq ~top-row '(1 2 3))

(setq ~bottom-row '(7 8 9))

(setq ~left-column '(1 4 7))

(setq ~right-column '(3 6 9))

(setq ~row-size 3)

(setq ~goal '((1 2 3 4 5 6 7 8 9) 9))

(setq ~board-size 9)

(setq ~table-size 362880)
```

```
(setq ~max-elements 362880)

(setq ~debug nil)

(load 'table.l)

(init-table ~table-size)

(def get-all-positions
  (lambda (goal)
    (let [open-list options new-position test-index (elements 0) (steps-to-goal
                                                                   0)]
      (setq open-list (tconc nil (list (pack-board goal) steps-to-goal)))
      (put-table (state-to-index (car goal)) steps-to-goal)
      (incr elements)
      (While
        [(and (car open-list) (< elements ~max-elements))
         (setq new-position (caar open-list))
         (cond
           [(eq (car open-list) (cdr open-list)) (setq open-list nil)]
           [t (setq open-list (rplaca open-list (cdar open-list)))])
         (setq options (next-states (unpack-board (car new-position))))
         (setq steps-to-goal (add1 (cadr new-position)))
         (let [configuration index]
           (While
             [options
              (setq configuration (Pop options))
              (setq index (state-to-index (car configuration)))
              (cond
                [(eq (get-table index) -1)
                 (put-table index steps-to-goal)
                 (incr elements)
                 (and (eq (remainder elements 1000) 0)
                      (msg (length (car open-list)) t))
                 (setq open-list
                       (tconc open-list
                              (list (pack-board configuration)
                                    steps-to-goal)))])]))]
        [(>= elements ~max-elements)
         (msg "too many elements" t t)
         ($prpr (car open-list))
         (return)])))

(def pack-board
  (lambda (state)
    (let [(val 0) (vec (car state))]
      (While
```

```
        [vec (setq val (+ (* 10 val) (Pop vec)))])
      val)))

(def unpack-board
  (lambda (val)
    (let [vec blank (i 9)]
      (While
        [(> i 0)
         (Push vec (remainder val 10))
         (setq val (quotient val 10))
         (cond
           [(eq (car vec) 9) (setq blank i)])
         (decr i)])
      (list vec blank))))

(def next-states
  (lambda (configuration)
    (let [end-conditions]
      (setq end-conditions (boundries configuration))
      (cond
        [(eq (length end-conditions) 0)
         (list (move-up configuration)
               (move-down configuration)
               (move-left configuration)
               (move-right configuration))]
        [(eq (length end-conditions) 1)
         (cond
           [(eq (car end-conditions) 1)
            { The blank is in left column}
            (list (move-up configuration)
                  (move-down configuration)
                  (move-right configuration))]
           [(eq (car end-conditions) 2)
            { The blank is in the right column}
            (list (move-up configuration)
                  (move-down configuration)
                  (move-left configuration))]
           [(eq (car end-conditions) 3)
            { The blank is in the top row}
            (list (move-down configuration)
                  (move-left configuration)
                  (move-right configuration))]
           [(eq (car end-conditions) 4)
            { The blank is in the bottom row}
            (list (move-up configuration)
                  (move-left configuration)
                  (move-right configuration))])]
        [t
```

```
        { Blank position satisfies 2 end conditions}
        (cond
          [(memq 1 end-conditions)
           { Blank in left col}
           (cond
             [(memq 3 end-conditions)
              { And in top row}
              (list (move-down configuration) (move-right configuration))]
             [t
              { And in bottom row}
              (list (move-up configuration) (move-right configuration))])]
          [t
           { Blank in right col.}
           (cond
             [(memq 3 end-conditions)
              { And in top row}
              (list (move-down configuration) (move-left configuration))]
             [t
              { And in bottom row}
              (list (move-up configuration) (move-left configuration))])])))))

(def boundries
  (lambda (configuration)
    (let [end-conditions (blank-position (cadr configuration))]
      (cond
        [(memq blank-position ~left-column) (Push end-conditions 1)])
      (cond
        [(memq blank-position ~right-column) (Push end-conditions 2)])
      (cond
        [(memq blank-position ~top-row) (Push end-conditions 3)])
      (cond
        [(memq blank-position ~bottom-row) (Push end-conditions 4)])
      end-conditions)))

(def move-up
  (lambda (configuration)
    (change configuration (- (cadr configuration) ~row-size))))

(def move-down
  (lambda (configuration)
    (change configuration (+ ~row-size (cadr configuration)))))

(def move-left
  (lambda (configuration)
    (change configuration (- (cadr configuration) 1))))

(def move-right
  (lambda (configuration)
```