The 15 Puzzle problems

Dr.N.Sairam & Dr.R.Seethalakshmi
School of Computing,
SASTRA Univeristy,
Thanjavur-613401.

Contents

1. The 15 Puzzle problem

During the last several years, there has been increasing interest in studying how humans solve combinatorial problems, such as the traveling salesman problem (TSP) and 15 puzzle problems and so on. In this section we will concentrate on 15 puzzle problem. The 15-puzzle has different-sized variants. The smallest size involves a board 2 X 3 and is called the 5-puzzle. The 8-puzzle involves a board 3X 3. The 35-puzzle involves aboard 6 X 6 (boards of other shapes—e.g., 4X 9—could also be used). The family of these puzzles will be called the *n-puzzle*, where *n* stands for the number of tiles. In all of the *n*-puzzles we used, the tiles in the goal state were ordered from left to right and from top to bottom, with an empty space located in the bottom right corner (previous studies of the 8-puzzle have usually used a goal state in which the empty space was in the center and the eight tiles were ordered around the boundary of the 3X 3 board). Previous research on the *n*-puzzle has concentrated on the 8-puzzle . It is known that the family of *n* puzzles belongs to the class of *NP* complete problems, which means that the number of paths grows exponentially with the number of tiles and that finding the shortest path from the start to the goal may require performing an exhaustive search.

Let us consider an example of 15 puzzle. We are defined with the start state and goal state as shown in the figure below.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Figure 1 Start State

1	2	3	4
5	6	7	8
7	10	11	12
13	14	15	

Figure 2 Goal State

1.1 Solution

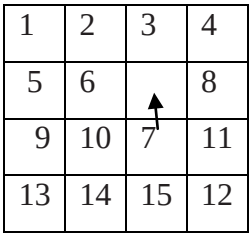

Figure 3 Step1

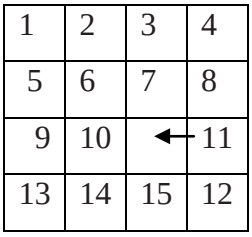

Figure 4 Step2

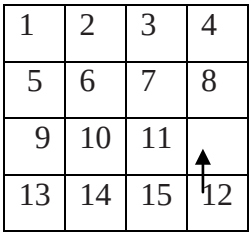

Figure 5 step3

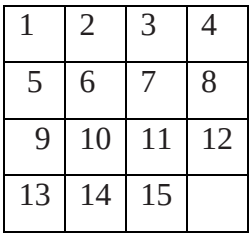

Figure 6 Target

The 15 puzzle can have any start state. First the start state is evaluated to check if it can lead to a goal state. Once this thing is verified, next we explore the state space to find the solution. There may be more than one solution to a given initial state. Among this we have to find the minimum distance as the heuristic. The iterative deepening A* algorithm is used to find the solution in the state space.

Depth First Iterative Deepening A*

Of the brute-force searches, depth-first iterative-deepening A*(DFIDA*) is the most practical, because it combines breadth-first optimality with the low space complexity of depth-first search. Its basic idea is as simple as conducting a series of independent depth-first (backtracking) searches, each with the look-ahead horizon extended by an additional tree level. With the iterative approach, DFIDA* is guaranteed to find the shortest solution path, just as a breadth-first search would.
But in contrast to the latter, DFIDA* needs negligible memory space. Its space complexity grows only linearly with the search depth.

```
algorithm Iterative Deepening;
begin
        bound := h(root);          //f initial bound is heuristic estimate g
        repeat
                bound := DepthFirstSearch (root; bound);          //f perform iterative-deepening
        DFS g
        until solved;
end.

function DepthFirstSearch (n, bound): integer;              //f returns next cost bound g
begin
        if h(n) = 0 then begin
                        solved := true; return (0);                          //f found a
                        solution: return cost g
                end;
                new bound := 1;
                for each successor ni of n do begin
                        if c(n; ni) + h(ni) _ bound then
                                b := c(n; ni) + DepthFirstSearch (ni; bound □c(n;
                        ni));                       //f search deeper g
                        else
                                b := c(n; ni) + h(ni); f cuto_ g
                                if solved then return (b);
                                        new bound := min (new bound; b);
                                        //f compute next iteration's bound g
                                end;
                        return (new bound);                                  //f return next
                iteration's bound g
end;
```

Iterative-Deepening A*, IDA* for short, performs a series of cost-bounded depth-first searches with successively increased cost-thresholds. The total cost f(n) of a node n is made up of g(n), the cost already spent in reaching that node, plus h(n), the estimated cost of the path to the nearest goal. At each iteration, IDA* does the search, cutting off all

nodes that exceed a fixed cost bound. At the beginning, the cost bound is set to the heuristic estimate of the initial state,
h(root). Then, for each iteration, the bound is increased to the minimum path value that exceeds the previous bound.

The algorithm consists of a main Iterative Deepening routine that sets up the cost bounds for the single iterations, and a Depth First Search function, that actually does the search. The maximum search depth is controlled by the parameter bound. When the estimated solution costc(n; ni) + h(ni) of a path going from node n via successor ni to a (yet unknown) goal node does not exceed the current bound, the search is deepened by recursively calling Depth First Search. Otherwise, subtree ni is cut off and the node expansion continues with the next successor ni+1.Of all path values that exceed the current bound, the smallest is used as a cost bound for the next iteration. It is computed by recursively backing up the cost values of all subtrees originating in the current node and storing the minimum value in the variable new bound. Note, that these backed-up values are revised cost bounds, which are usually higher { and thus more valuable { than a direct heuristic estimate. In the simple IDA* algorithm shown in above the revised cost bounds are only used to determine the cost threshold for the next iteration.

In conjunction with a transposition table they can also serve to increase the cut offs. With an admissible heuristic estimate function (i.e. one that never overestimates), IDA* is guaranteed to find the shortest solution path. Moreover, it has been proved that IDA* obeys the same asymptotic branching factor asA*, if the number of nodes grows exponentially with the solution depth. This growth rate is called the heuristic branching factor bh. On the average IDA* requires bh=(bh □ 1) times as many operations as A*. While the search overhead diminishes with increasing bh (e.g., 11% overhead at bh = 10,1% at bh = 100), IDA* benefits from the elimination of unnecessary node re examinations in the shallow tree parts (all iterations before the last).