Artificial Intelligence CS 531: Programming Assignment 2

Roli Khanna and Vivswan Shitole

Oregon State University

Abstract. This is the documentation of the second programming assignment in CS 531 Artificial Intelligence (Winter 2019) class. We introduce the problem with the search algorithms involved and the performance measures used to compare the algorithms. This is followed by a formal description of the two search algorithms used: Iterative Deepening A* and Recursive Best First Search. Next we describe the experimental setup involving a description of the 15-puzzle problem, its representation as a state-space and the heuristic functions used. Then we present the obtained results and plots. Finally we conclude with a discussion of the follow-up questions asked in the problem statement.

Keywords: Artificial Intelligence · 15-puzzle · State Space Search · Iterative Deepening A* · Recursive Best First Search.

1 Introduction

A summary of the problem statement is as follows: We have to solve the 15-puzzle game (generalization of the 8-puzzle game from a 3X3 grid to a 4X4 grid) using two search algorithms: Iterative Deepening A* and Recursive Best First Search. Since these are state-space search algorithms, we have to first formulate the 15-puzzle problem as an implicit state-space graph and then search on the dynamically generated graph using the two algorithms.

Moreover, we use two different heuristic functions to generate the estimated heuristic values for the states of the game generated under each search algorithm. We compare both the algorithms for each of the heuristics by reporting the following metrics over 50 different initial game states:

- Average time taken to solve a game instance.
- Optimal solution length (length of the action sequence) on average.
- Average number of nodes (states) searched before solving a game instance.
- Fraction of problems solved (thresholded by maximum number of nodes searched = 100000).
- Fraction of the total time spent on calculating the heuristic values.

2 Description of Search Algorithms

2.1 Iterative Deepening A*

Iterative Deepening A* combines the standard A* search algorithm with iterative deepening based on the f-values. Hence, we begin with the initial f-limit as the f-value for the initial state. We maintain a priority queue in order to visit states in increasing order of f-values. If we find the goal state within the current f-limit, we return the action sequence so far as the solution. In case the f-limit is exceeded by the child states, we update the new f-limit as the minimum of the f-values of all the child states encountered and start over. This approach guarantees consistence and optimality as long as the heuristic function used is admissible.

The Iterative Deepening A* algorithm is as follows:

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f- COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f- COST(root)
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
  inputs: node, a node
          f-limit, the current f- COST limit
  static: next-f, the f- COST limit for the next contour, initially ∞

  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

2.2 Recursive Best First Search

Recursive best-first search (RBFS) is a recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up valuethe best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten

subtree and can therefore decide whether its worth re-expanding the subtree at some later time. RBFS is consistent and optimal as long as the heuristic function used is admissible.

The Recursive Best First Search algorithm is as follows:

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

3 Description of Heuristic Functions

3.1 Manhattan Distance

This heuristic gives the sum of L1-norms for each of the 15 numbers in the current state of the board from their respective positions in the goal state of the board. Hence the heuristic function is calculated as:

$$h(s) = \sum_{n=1}^{15} (\mid n(s)_x - n(g)_x \mid + \mid n(s)_y - n(g)_y \mid)$$

where,

$n(s)_x$ denotes x co-ordinate of the number n in the current board state s
$n(g)_x$ denotes x co-ordinate of the number n in the goal board state g

This is the first heuristic that we have used for estimating the h-values. It is an admissible heuristic as it is a solution to a relaxed version of the original problem where we assume that the number blocks can move freely, without being restricted by other blocks in the way.

3.2 Frobenius Norm

The Frobenius norm is one of the standard matrix norms, also known as the effective Eucledean norm for a matrix. The Frobenius norm of a matrix A is given as:

$$|| A ||_F = \sqrt{Tr\,(AA^H)}$$

We build over this matrix norm to form a custom admissible norm as follows:

1. Let the current state board matrix be S. Let the goal state board matrix be G. We form a difference matrix D as:
$$D = S - G$$
2. Calculate the Frobenius norm of the difference matrix D:
$$K = || D ||_F = \sqrt{Tr\,(DD^T)}$$
3. Restrict the Frobenius norm K between [0, 1] using the sigmoid function:
$$L = \frac{1}{1+\exp^{-K}}$$
4. Calculate the Manhattan distance of the original current state matrix S:
$$M = \sum_{n=1}^{15} (\mid n(S)_x - n(G)_x \mid + \mid n(S)_y - n(G)_y \mid)$$
5. The custom heuristic H is then given as:
$$H = L + M$$

Hence, our custom heuristic is the sum of the Manhattan distance with the sigmoid of the Frobenius norm of the difference matrix. Taking the sigmoid in step-3 is the key step to make the custom heuristic admissible since it restricts the value between [0, 1].

4 Experiments and Inferences

4.1 Formulation of 15-puzzle as implicit State-Space Graph

To enable search, we formulated the 15-puzzle game as an implicit state space graph. The board with 16 cells was represented by a 4X4 matrix. All the cells are assigned numbers except for the one empty cell. Each unique arrangement of these 16 entities is considered as a separate state node in the state-space graph. The action space then consists of four actions: {left, right, up, down} denoting the movement of the empty cell. Some of these actions become invalid when the empty cell is at the board boundary. Thus we have a maximum branching factor of 4, one for each action, leading to the next state.

4.2 Setting Up Episodes with Different Initial States

We need to generate results for many game-play episodes to have definitive results from which we can draw substantial inference. Each of these episodes should preferably begin with a different initial state so that our inference is over a broad diversity of game-plays.

We use the Scramble(m) function to generate random initial states. Each of these states is generated by a random m-move sequence from the final goal state. We generate 10 different initial states for each m and m itself is varied over the set of values {10, 20, 30, 40, 50}. Each of the two search algorithms are then used to solve the same randomly generated initial board, using one of the heuristic functions. Hence we have a valid comparison of the performance of the two search algorithms as well as the two heuristic functions over a same initial state.

4.3 Plots and Inference

Iterative Deepening A* In this section, we provide the plots obtained for Iterative Deepening A* search using the Manhattan distance heuristic as well as the our custom heuristic. We also discuss the inferences from the plots.

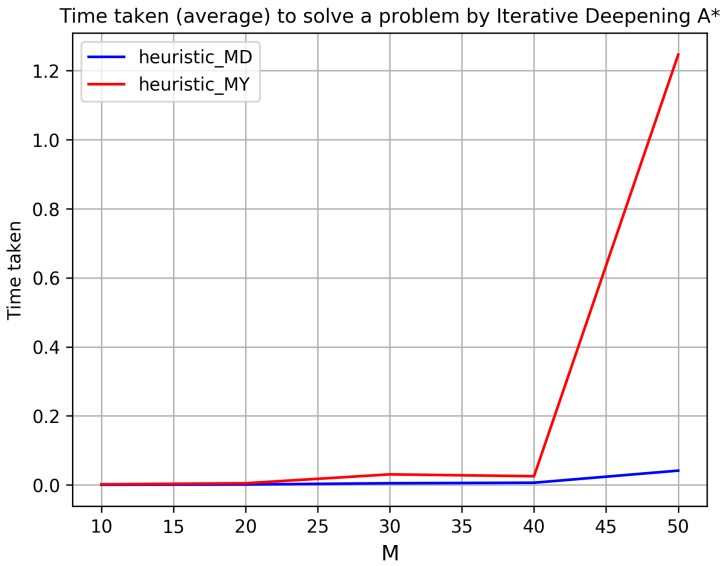

Fig. 1: Average time taken to solve a problem by Iterative Deepening A* algorithm.

Fig.1 compares the average time taken by the IDA* algorithm to solve a problem using the Manhattan distance (MD) heuristic and the custom (MY) heuristic for different values of M (corresponding to different initial states). We observe that the time taken to solve a problem increases with increase in the initial states perturbations M. This is expected since the initial state is farther (more perturbed) from the goal state as M increases. The more interesting observation is that the time taken to solve the problem using MY heuristic increases

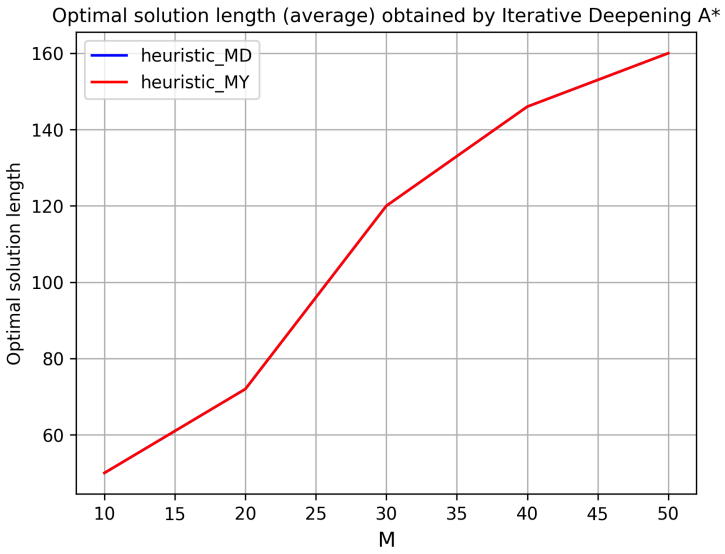

Fig. 2: Optimal solution lengths obtained by Iterative Deepening A* algorithm.

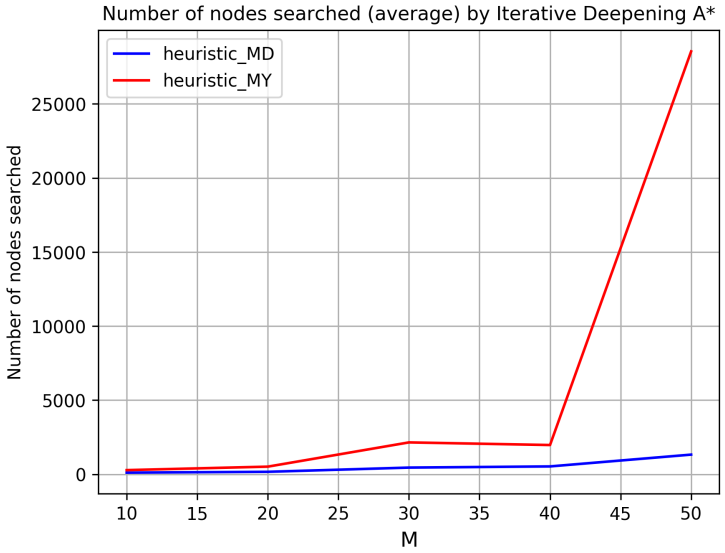

Fig. 3: Number of nodes searched on average by Iterative Deepening A* algorithm.

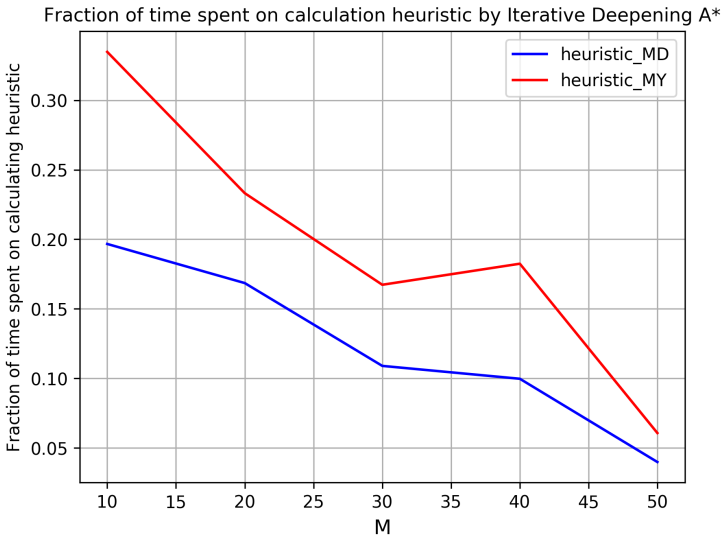

Fig. 4: Fraction of the total time spent on calculating the heuristic function by Iterative Deepening A* algorithm.

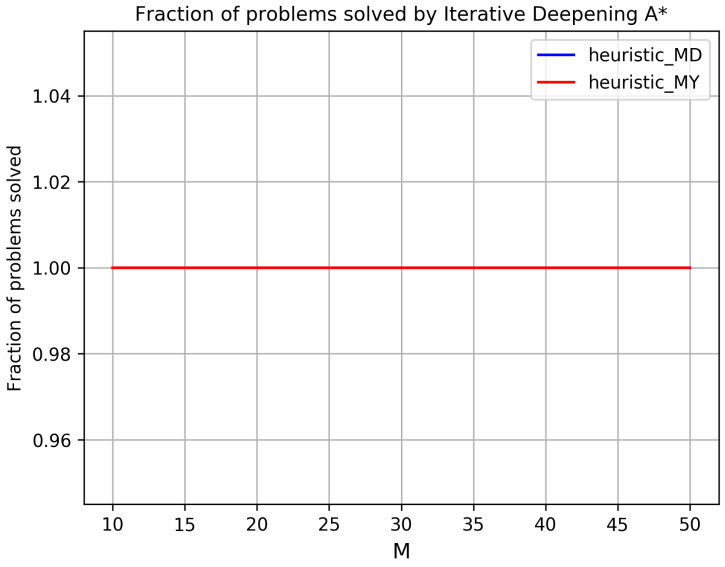

Fig. 5: Fraction of games solved successfully by Iterative Deepening A* algorithm within the computation limit imposed.

exponentially compared to that using MD heuristic for the same problem. This implies MD heuristic is a much better heuristic than MY, as far as solving time is concerned. We believe this is because MD is closer to the true (unknown) heuristic h* than MY.

Fig.2 compares the optimal solution lengths obtained by the IDA* algorithm on average using MD and MY heuristics for different values of M (corresponding to different initial states). We observe that the optimal solution lengths are exactly the same for both MD and MY heuristics. This implies both the heuristics find the same optimal solutions, implying that both of them are admissible. We do not have a preference for one of the heuristics as far as the optimal solution lengths are concerned.

Fig.3 compares the average number of nodes searched by the IDA* algorithm to solve a problem using MD and MY heuristics for different values of M (corresponding to different initial states). We observe that the average number of nodes searched to solve a problem increases with increase in the initial states perturbations M. This is expected since the initial state is farther (more perturbed) from the goal state as M increases. Again, we observe that the number of nodes searched to solve the problem using MY heuristic increases exponentially compared to that using MD heuristic for the same problem. This implies MD heuristic is a much better heuristic than MY, as far as number of nodes searched to solve the problem are concerned. This reinforces that MD is closer to the true (unknown) heuristic h* than MY.

Fig.4 compares the fraction of the total solving time spent on calculating the heuristic values by the IDA* algorithm for MD and MY heuristics over different values of M (corresponding to different initial states). We observe that the fraction decreases with increase in M. This implies as the initial state becomes more perturbed from the goal state, it gets pushed deeper into the state-space graph. Hence, a greater fraction of the total solving time is spent on traversing the graph and searching the goal node than calculating the heuristic values for all nodes. Moreover, we observe that calculating MY heuristic takes a larger chunk of the total time than MD. This implies MD heuristic values can be calculated more efficiently than MY values, thus yielding a preference for MD heuristic function.

Fig.5 compares the fraction of the total problems solved by the IDA* algorithm using MD and MY heuristics over different values of M (corresponding to different initial states). We observe that the value is constant at 1.0, implying all of the problems were solved by both the heuristics within the imposed computation (search) limit of MAX NODES SEARCHED = 100000. There is no preference for any of the two heuristics from this perspective.

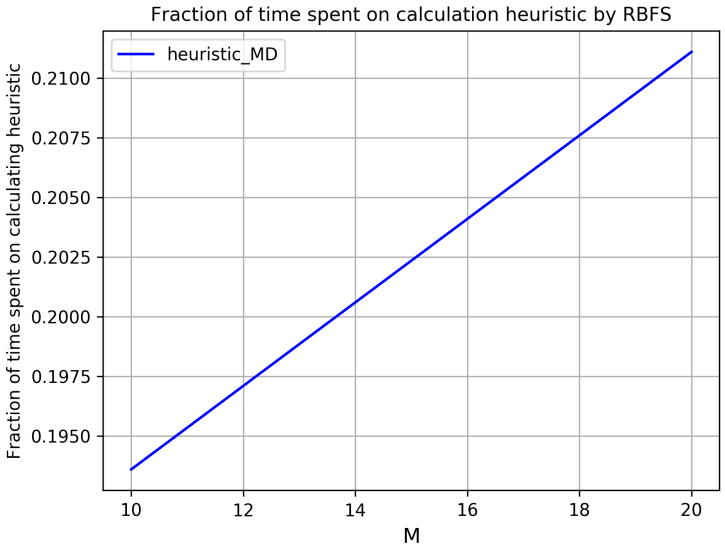

Fig. 6: Time taken for execution on average by the RBFS algorithm.

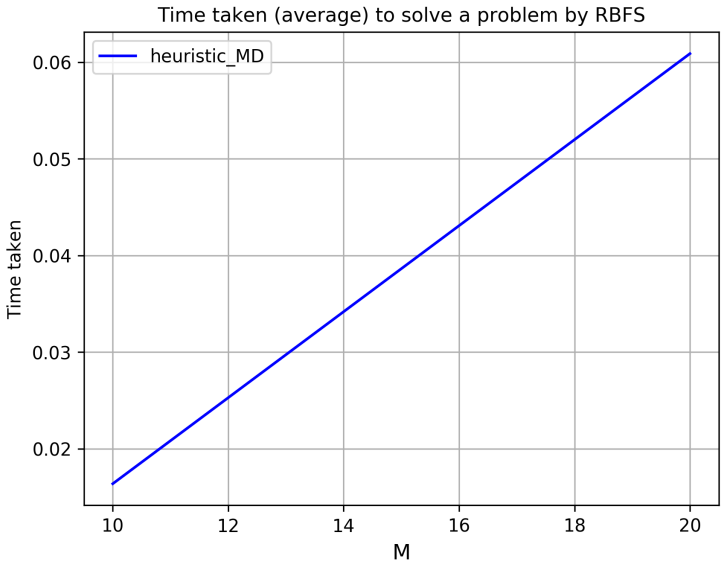

Fig. 7: Time taken by the RBFS algorithm.

Recursive Best First Search

5 Conclusion and Discussion

We developed the state-space formulation of the 15-puzzle problem and compared the performance of two search algorithms: iterative deepening A* and Recursive Best First Search using two heuristic functions Manhattan Distance (MD) and a custom heuristic (MY). We obtained the results over 50 different initializations of the game and drew important inference from them. Here, we discuss the answers of some of the questions in the problem statement:

1. *Is there a clear preference ordering among the heuristics you tested considering the number of nodes searched and the total CPU time taken to solve the problems for the two algorithms?*

   From the plots for average solving time and number of nodes searched, along with their inferred discussion, we conclude that MD heuristic has a clear preference over MY heuristic.

2. *Can a small sacrifice in optimality give a large reduction in the number of nodes expanded? What about CPU time?*

   Yes. For e.g. using DFS instead of Iterative Deepening A* would sacrifice the optimality guarantee but reduce the number of nodes expanded, considering only the cases where DFS does reach the solution. The CPU time, if counted only for the cases where both the algorithms reach a solution, will decrease since its directly correlated with the number of nodes expanded. However, the total CPU time for sub-optimal algorithm may be exponentially higher than the optimal algorithm as it may not always find a solution.

3. *Is the time spent on evaluating the heuristic a significant fraction of the total problem-solving time for any heuristic and algorithm you tested?*

   As we note from the plots depicting the fraction of the total solving time spent on calculating the heuristic values, the maximum values for the MD and MY heuristics are around 0.35 and 0.2 respectively. Hence, in both the cases, the time spent on evaluating the heuristic is not a significant fraction of the total problem-solving time.

4. *How did you come up with your heuristic evaluation function?*

   Since the heuristic values are distance metrics, using a matrix norm such as the Frobenius norm seemed to be a good idea to us. As the heuristic should measure the distance between the current state and the goal state, we decided to take the Frobenius norm of the difference matrix of the current

state and the goal state. The next requirement was to make the heuristic function admissible. Hence, we bounded the value of the Frobenius norm of the difference matrix by feeding it to the sigmoid function. Finally we added it to the Manhattan distance heuristic to obtain the final custom heuristic that benifits from both the Frobenius norm and the Manhattan distance.

5. *How do the two algorithms compare in the amount of search involved and the cpu-time?*

   As we observe from the plots of average time taken and average number of nodes searched to solve the problem for IDA* are much higher than RBFS. Thus RBFS is better than IDA*.