Report for Programming Assignment 2 (Fifteen Puzzle)

Matthew Callahan **Suphalerk Lortaraprasert**

Abstract

We consider the performance of two algorithms, A* search and rbfs, on finding solutions to a generic fifteen puzzle. We compared their performance with respect to computation time and optimality of solution for various levels of scrambling of the board and with respect to two different heuristics.

1 Introduction

In this paper, we examine the performance of two search algorithms, Compassion A* and RBFS, using a 4x4 puzzle as a representation. The environment is randomized to ensure optimal exploration. To enhance efficiency, we introduce a random algorithm designed to avoid repeating the same steps.

2 Environment Descriptions

We consider the setting of the fifteen puzzle. In this setting, a four by four grid contains fifteen(15) tiles, where the tiles can be slid to move the empty square around. The tiles are each numbered and the goal of the program is to find a sequence of moves that will return the tiles to their correct order after being scrambled in the minimum moves with the minimum amount of calculation time.

2.1 Heuristic function

We used the Manhattan distance as one of our heuristics. For this, we considered each tile to have a target location, and counted the total number of squares each tile must be moved to reach that location. This is equivalent to assuming each tile can be moved through other tiles.

We also used a heuristic suggested by Michael Kim, on his blog at https://michael.kim/blog/puzzle, where the Manhattan distance is augmented by an addition of two moves for each location that has two tiles next to each other that need to be directly swapped. This is because it takes at least three moves to swap two tiles. This is no longer an admissible heuristic because it assumes that each swap must happen independently.

This is not quite the same as what is reccomended by Mr. Kim, since he does not mention anythin about what to do with vertical swaps. We treated vertical swaps differently because the correct offset between vertical tiles is 4 instead of 1.

2.2 A* search

We implement the A* search algorithm for finding the paths between initial states to a given goal state in the problem of puzzle 4x4. It is a variant of the uniform cost search by using a heuristic function for each node in addition to path costs.

Preprint. Under review.

Algorithm 1 Programmatic Description of A* search

```
openlist = [Node(nodeState, 0, nodeState.manhdist())] // Start with the initial state
closedlist = []
totalHeuristic = 0 // keep running total of heuristic time
while openlist is not empty: do
    currentnode = node with minimum fscore in openlist
    currentnode.state.checkcanmove()
    remove currentnode from openlist
    add currentnode to closedlist
    if  currentnode.heuristic is 0 then
        return currentnode, length of closedlist, totalHeuristic // Return the goal node and the number
of nodes expanded
    for move in ["up", "down", "left", "right"] do
        successorstate = copy of currentnode.state
        cannotmove = currentnode.state.checkcanmove()
        if move in cannotmove: then
            continue
        if move is "up": then
            successorstate.moveUp()
        if move is "down": then
            successorstate.moveDown()
        if move is "left": then
            successorstate.moveLeft()
        if move is "right": then
            successorstate.moveRight()
        cost = currentnode.cost + 1 // Assume uniform cost for each move
        heuristicTic = time.perfcounter()
        if  funType is "other": then
            heuristic = successorstate.otherHeuristic()
        else
            heuristic = successorstate.manhdist()
        heuristicToc = time.perfcounter()
        totalHeuristic += heuristicToc - heuristicTic
        successornode = Node(successorstate, cost, heuristic, currentnode)
        if successornode not in closedlist: then // This would prefer a better equality
            add successornode to openlist
    return None
```

2.3 Recursive Best-First Search (RBFS)

Recursive Best-First Search (RBFS) operates by maintaining an f-limit variable, tracking the f-value of the best alternative path from any ancestor node to the current node. This f-limit guides the algorithm in determining which subtree of the problem tree to explore, balancing between the current path and the best alternative path. RBFS dynamically updates the f-values of nodes during recursion unwinding to ensure search functionality, thereby allowing consideration of forgotten subtrees in future exploration.

RBFS offers the advantage of requiring less memory compared to A* search, as it utilizes linear space. Unlike A*, which stores all explored nodes, RBFS only retains "relevant" nodes in memory. However, RBFS may expand more nodes than A* due to redundancy. Since RBFS doesn't store all explored nodes, it can potentially revisit and re-expand the same nodes, leading to increased computation time.

```
if node.state isgoal() then
    return node, flimit
successors = []
for move in ["up", "down", "left", "right"] do
    if move is not in node.state.checkcanmove() then
        successorstate = copy of node.state
        performmove(move, successorstate) // Execute the move on the state
        cost = node.cost + 1
        heuristic = calculateheuristic(successorstate) // Calculate heuristic value for the successor state
        successornode = create Node with successorstate, cost, heuristic, and parent node
        add successornode to successors list
if successors is empty then
    return None, infinity
while true do
    sort successors by fscore
    best = first node in successors
    if best.fscore > flimit then
        return None, best.fscore
    alternative = second node's fscore if successors has more than one node else infinity
    result, best.fscore = RBFS(best, min(flimit, alternative))
    if result is not None then
        return result, best.fscore
```

3 Experimental setup

To evaluate the performance of the A* search and Recursive Best-First Search (RBFS) algorithms in solving the fifteen puzzle problem, we conducted a series of experiments under controlled conditions. Here is an overview of our experimental setup:

Puzzle Generation:

We generated puzzles of varying sizes, ranging from 10 to 50, to assess the scalability of the algorithms. Each puzzle was initialized with a random configuration to ensure unbiased testing.

Algorithm Configuration:

We implemented the A* search algorithm and RBFS algorithm in Python to solve the generated puzzles. For A* search, we utilized two different heuristic functions: Manhattan distance and the heuristic suggested by Michael Kim, as described in Section 2.1.

Performance Metrics:

We measured the average solving time for each algorithm and heuristic combination. Additionally, we recorded the average number of nodes expanded by each algorithm during the search process. The length of the solution path was also monitored to evaluate the optimality of the solutions produced by the algorithms.

Replicability:

To ensure the replicability of our experiments, we ran each algorithm and heuristic combination multiple times on different puzzle instances. We recorded the average values of solving time, nodes expanded, and solution length across multiple trials for each puzzle size and algorithm configuration.

By systematically varying the puzzle sizes and algorithm configurations while maintaining consistency in experimental conditions, we aimed to obtain reliable results for comparing the performance of A* search and RBFS algorithms in solving the fifteen puzzle problem.

4 Results

The results of our experimentation are summarized in Table 1. We conducted trials on puzzles of varying sizes, ranging from 10 to 50, using both A* search with heuristic and Recursive Best-First Search (RBFS).

M :	10	20	30	40	50
Avg. Time (A*)	0.00038433	0.0008775	0.00324463	0.00929986	0.02746816
Avg. Time (Heuristic A*)	0.00014227	0.00028497	0.00065334	0.00111751	0.00161616
Avg. Nodes (A*)	4.2	7.7	17.3	29.1	42.2
Avg. Length (A*)	2.9	5.9	9.4	10.1	12
Avg. Time (RBFS)	0.00052958	0.0012561	0.00679943	0.03392225	0.85811134
Avg. Length (RBFS)	2.9	5.9	9.4	10.1	12

Table 1: Comparison of A* Search and RBFS on Different Puzzle Sizes

The results demonstrate that A* search consistently outperforms Recursive Best-First Search (RBFS) across all puzzle sizes tested. A* search exhibits shorter average solving times and requires fewer nodes to find a solution compared to RBFS. Additionally, while both algorithms produce solutions of comparable lengths, A* search achieves these results more efficiently, making it a better choice for solving the 4x4 puzzle problem.

5 Discussion

5.1 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?

Yes in our result, A* generally surpasses RBFS in terms of time efficiency owing to its methodical and informed traversal of the search space. A* relies on heuristic estimates and optimal cost considerations to guide its exploration, resulting in a more effective approach compared to RBFS.

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

The non-admissible heuristic trades off optimality for efficiency, as it doesn't always produce the optimal solution. However, this compromise in optimality can result in a significant reduction in the number of expanded nodes, leading to decreased overall CPU time. For instance, while an admissible heuristic might underestimate the true cost, causing a large number of node expansions, a non-admissible heuristic may overestimate the cost by only a small margin. Moreover, intentionally overestimating the cost can sometimes simplify the problem. Consequently, the use of a non-admissible heuristic can lead to a substantial decrease in node expansions and, consequently, reduced CPU time compared to an admissible heuristic. An illustrative case is our second non-admissible heuristic, where A* demonstrates exceptional performance.

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

5.4 How did you come up with your heuristic evaluation function?

We looked to the literature for suggestions.

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

Amount of Search Involved:

A*: It is a complete, optimal search algorithm. A* expands nodes in a best-first manner, prioritizing nodes based on a combination of the cost to reach a node from the start and a heuristic estimate of the cost to get from the node to the goal. It guarantees finding the optimal path if a solution exists.

RBFS: It is also a complete, optimal algorithm. RBFS expands nodes in a best-first manner similar to A*, but it utilizes recursion to handle larger search spaces efficiently. RBFS can be seen as a memory-bounded version of A* because it stores only the best path and the alternative paths that are currently being explored.

CPU Time:

A*: The CPU time required by A* depends on several factors, including the size of the search space, the quality of the heuristic function, and the specific implementation details. In general, A* can be quite efficient, especially when a good heuristic is available and the search space is not excessively large.

RBFS: The CPU time of RBFS is typically competitive with A* for smaller search spaces. However, as the search space grows larger, RBFS may outperform A* in terms of memory efficiency but could require more CPU time due to the overhead of recursive function calls and backtracking.

In summary, both A* and RBFS are effective search algorithms with their strengths and weaknesses. A* is well-suited for problems where memory is not a significant constraint and a good heuristic function is available. RBFS, on the other hand, can be more memory-efficient but may require more CPU time for larger search spaces due to its recursive nature. The choice between the two depends on the specific characteristics of the problem at hand and the available computational resources.