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

4 Results

5 Discussion

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?

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.

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?

4. How did you come up with your heuristic evaluation function? We looked to the literature for suggestions.

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