Dylan Dunsheath HW7: 15-Puzzle

The requirements for this specific program was to implement A\* Search Algorithm for a 15-Puzzle that approximates the shortest path from the “initial state” to the “goal state” (where the values **should be** from 1-15 and a 0 or “blank space/tile” in the puzzle). To start off the program, we have 3 standard C libraries that were implemented or included: stdio.h; allowing us to use printf, stdlib.h; specifically for memory allocation, and string.h; allowing us to compare two strings (also has other use cases in many programs). There are also four different macros: N which is used with NxN (N\*N) for the puzzle dimensions, and the macros TRUE (1) and FALSE (0).

The second part of the program that is significant is the structure (called node). This is a fundamental part of the data structure of this program, as it represents a state of the puzzle. Its members are the following: a two dimensional array called tiles that equates to a N x N identity matrix; as it represents an array with equal rows and columns that hold the values from 0 – 15 (0 is technically not existing, as it’s more of a blank space). There is also naturally the f, g, and h values for pathfinding. F is an “estimation of the total cost from the path from the initial through n to the goal” (or the sum of g and h). G represents the actual cost and h is an estimated cost to each the goal from N node. There is also short zero\_row and short zero\_column which helps us where the empty space is located in the array or puzzle. There are also two pointers; next and parent; where next is used to link nodes together in a linked list structure and points to the next node and parent which is used to ppoint to the parent node in the search tree and trace back the solution path once the goal state is reached.

Some global variables are the goal\_row and goal\_column that’s 1 dimension of size 16 (Given N = 4). These arrays will be used to compare the positions of the tiles in the current state with the goal state during the A\* search. There are four pointers as well that are a pointer to the structure” struct node: two being start and goal; which are used to keep track of the state and goal state of the puzzle. Then there. The next two are open and closed; which are explicitly initialized to NULL. Open is used to that have candidates for expansion or yet to be explored while closed are nodes that were already explored. Lastly, there is struct node \*succ\_nodes[4]: showing an array of pointers to structure called node with a size of 4 that are used to store successor nodes during expansion of the parent. There is a potential successor within each element when moving the blank tile right, left, up, or down.

Our first function: print\_a\_node is responsible for printing the contents of the puzzle as a 4x4 array. It takes a pointer to a struct node as input and is denoted by pnode. Within the body of this function, there is a nested loop\ that uses i and j as the loop variables to print the rows (outer) and columns (inner) and does printf(“%2d ”, pnode->tiles[i][j]) to print the value of the tile at these positions; where 2d specifies a width of at least two characters to ensure consistency in the output. We also ensure a newline after the inner loop is executed and when both loops are done as well.

The second function, initialize is essentially responsible for initializing the grid and setting up both the initial and goal states and takes the pointer to an array of strigns from argv as an argument. This (supposedly, as it terminates if any error is found when passing arguments) to the puzzle grid. One important thing that takes place is the dynamically allocated memory for a new ‘struct node’ and that’s assigned to the variable pnode. After the dynamic memory allocation is completed, the nested loop with j and k iterate over the NxN puzzle grid and extracts the integer value of the puzzle from the command-line using atoi… the code is tile = atoi(argv[index++]) where index starts at 1 (assigned this value). The next line then assigns this value to the corresponding position in the grid from the command-line. It's also important to keep track of where the 0 or empty tile is located: so if the tile is a 0, we then use the j and k to assign it to the zero\_row and zero\_column values by also using the -> operator since it is in the structure. The rest of the code in essentially initializing the node attributes: where f, g, and h are 0 and the pointers are NULL. We then set up the start / initial state and print this to the screen, set up the goal state with dynamic memory allocation and set up the goal state by population this puzzle grid with values 1-15 and the last tile as 0 or empty. The attributes of f, g, h which are attributes to the goal node are 0 and next points to nothing. We then assign the newly assigned goal node to the goal pointer and print this goal state. We then need to return the start pointer, the start state of the puzzle.

The free\_memory function is arguably the most simple explanation. First, we free the start and goal nodes of the program (since these were dynamically allocated within initialize). However, deallocating for open and closed is a bit more complicated, we need to iterate through each node in these lists (using a while loop), store the next node’s pointer next to prevent loss of memory, free the memory allocated by the current node called temp and update temp. Freeing the succ\_nodes simply involved looping over it and freeing the current index. I did this function to prevent any memory leaks in my program.

The merge\_to\_open is a function that essentially merges successor nodes into the open list based on their f value. We iterate over the succ\_nodes and check if it is NULL. If it is, we continue. We then allocate memory for a node called noInsert and copy the puzzle configuration and anything else that may be deemed relevant from succ\_nodes: including the f, g, and h values along with the zero row and column and the parent. Then, I check if the open list is empty and if it is, set it to the newly created node toInsert. If not, I traverse the open list to then find what the correct position to insert the new node is based on the f value and insert it in the open list to maintain the ascending order of f. If the new node has the lowest ‘f’ value among all the nodes in this list, it becomes the new head. If it’s higher than all nodes in the list, it’s inserted at the end of the open list. Essentially, this ensures all succ\_nodes are efficiently inserted into the list based on the f-value and maintains the open list in ascending order of f to make an algorithm that explores nodes with a lower estimated cost first and then potentially leading to faster searches toward goal state.

The swap function isn’t all that complicated: it’s essentially responsible for swapping the values of two tiles in the puzzle where we have row1 and column1 for one and row2 and column2 for the other. We essentially store the value at row1, column1 to a temp variable, the value at row2, column2 to row1, column1 and then temp to row2, column 2.

The manhattanDist() function essentially is a heuristic approach and is the sum of all absolute values of differences in the goal’s x and y coordinates and the current x and y coordinate in respect to us.

The min function is helpful in determining the lowest of two values: and b to find what the lower ‘h’ is when we update in update\_fgh function from the n to goal. The goal in A\* is to find the most optimal solution and this allows us to do so.

The most ideal thing is to have an update\_fgh function and update the corresponding values to truly see if what we are doing is the most optimal solution: since g is the cost TO the current node, h (the heuristic value) which is calculated with two different heuristic function: h1 being the number of misplaced tiles compared to the goal state and h2 being the Manhattan distance from its tile to the current position to goal. It uses both to find the minimum value of h1 and h2 as the approach for h (which are passed to min). It then uses g and h and sums these up and assigns it to f.

The next four function are essentially just movement functions that moves the blank tile in the specified direction.

The expand function is VERY crucial in the A\* algorithm and is responsible for the generation of successor nodes from a given node ‘selected’ and applies all possible valid moves that can be made to the tile and creates a new puzzle configuration called a successor node. Each of the successor nodes is the same configuration BUT with 0 or blank tile moved on a specific direction. The important part is when then calculate the cost and consider factors like how many moves were made (based on g), how close it is to the goal state (h), and the total sum (f).

Nodes\_same() essentially compare if two nodes are the same

Filter() is used to filter out any duplicate successor nodes

countInversions and isSolvable are functions that are used to see if a 15 puzzle is solvable or not by counting the number of inversions by flattening into a 1D array and checks if two tiles form an inversion (a tile with a higher value precedes a lower with a lower value). If it is found, it increments the counter for it. isSolvable uses the countInversions in order to not waste time calculating an impossible puzzle. In the isSolvable, we call countInversions to get the total number and retrieve the blankRow (where the 0) is. We then check if the grid length is even (it will ALWAYS be when the program is ran successfully with correct arguments) and the scenario where either

1. the blank row is on an EVEN row and inversions is ODD…
2. Blank row is on an ODD row and inversions is EVEN

* This is determined by my return statement

The hasDupes takes argc and argv as arguments and determines if there are any duplicate values…

The main function is what drives our program to do what we want. In my case, the function starts off with validating if the user passed EXACTLY 17 arguments to the program: the ./executable <Values from 0 - 15> and if not, it displays an error to the screen (via printf). The program also checks argv[1]…argv[argc – 1] values to see if they are within the range of 0 – 15. Finally, I checked with my hasDupes function if there were any duplicate values passed. If there are no errors, the program can run as intended.

After all these error validations, I declared several variables: iter, cnt, ret, I, pathlen, and index[N – 1] of type int. I also declared struct node \*copen, \*cp, and \*solution\_path; where solution\_path is initialized to null. I then do start = initialize(argv) which naturally initializes the puzzle and I then validate the validity of the puzzle: checking if it can be solved. If it can’t, based on the isSolvable and countInversions function, it throws that there is no solution to the puzzle and exits out of the program.

Then, the open list is initialized with the ‘start’ state. The program runs until one of two conditions are met: either the open list becomes empty or we found a solution [given the while (open ! = NULL)] where in each iteration, a node called copen is selected from the open list for expansion. IF the selected node matches the ‘goal’ state then a solution was found and the function constructs the solution path by tracing through the parent node sand it prints the solution path. Otherwise, it’s expanded and the successors are generated and added to the ‘open’ list after we filter using the filter function. We then merge the open with the closed list and the copen is moved to closed list. After the while loop is finished, we clear all the dynamic memory to prevent any potential errors or memory leaks.