**CPSC 481 – Term Project:**8-Puzzle Implementation Using

A\* Algorithm

**Name:** Mario Andrade

**Due Date:** April 22, 2015

**MW:** 4:00 – 5:15 PM

**Introduction:**

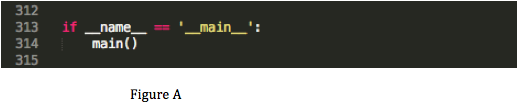
The 8-puzzle, a board with eight tiles and a blank spot that allows the player to move the tiles into the desired winning state, is one of the classic games that tends to be used so that people can have a better understanding of artificial intelligence. I implemented this puzzle for the term project because I wanted to become more familiar with the A\* algorithm and the heuristics that were provided to us in the textbook, as well as getting further practice in using algorithms to solve computer programming problems. The three heuristics that I implemented on this project were as follows: a heuristic that calculates the cost by counting how many tiles are out place, one that calculates the physical distance that a tile is currently positioned at to its goal state position, and one that uses the previous heuristic as a base and adds to it whenever two tiles would need to be reversed in order for them to be in their goal position. For this project, the programming language that I decided to use was Python. More specifically, the version that I used was 3.4.1. I chose Python because I had become very familiar in using it for the Algorithms class. I felt that I could program more easily in Python, than using C or C++, since Python has more preferable data structures than C++, when given a couple of weeks to get the program running. The risk of using Python, which I learned in the Algorithms class, is that when it comes to large amounts of data, it can perform at a slower speed than C because it is not as close to assembly language as C is. I also learned that with the proper algorithms, such as the in-place selection sort provided in this project could help alleviate some of the issues with Python being a higher-level language. I used the pseudocode provided by George Luger in his Artificial Intelligence textbook. Since, Python is a high-level language it makes it easier for pseudocode to be translated into functioning code than languages such as C. It also provides an easy way to understand what the code is doing by reading the code.

**Program Summary:**

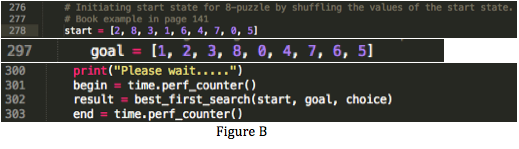
The goal of this program is to generate a solution to the given starting state, be it random or one of the static states provided either from the book or myself. The solution can be executed by the three heuristics I chose: tiles out of place, sum of distances out of place, and tile reversals. Each solution will go through a while-loop that takes the lowest-cost state from the list of open states and see if the state is the goal state. If not, it will generate that chosen state’s children and give them heuristic values based on which heuristic was chosen. Each child will be evaluated to see if they already exist in the open and closed states. If it does and it has a lower cost, the child removes the existing state from that list, and then the child gets added to the open list. Otherwise, it is a new state and it just adds itself to the open state. At the end of the iteration, the open state list gets sorted using an in-place selection sort and starts a new iteration of the while-loop. When a solution is found, the program returns to the main the list of closed states, which is the path the algorithm, took to find the goal state. This list will be printed out along with the depth and cost of each state and the how long it took to execute the algorithm, in seconds. The code is provided in the appendix of this paper, as well as in my GitHub page <https://github.com/Xoriam/A-star-Algorithm_Artificial-Intelligence> , to help follow along with the execution trace.

**Execution Trace:**

This program begins in the command line, where the user inputs the line *python3 8-puzzle.py* followed by an integer that will choose from one of the three heuristics. The program begins by importing the three libraries sys, random, and time. Sys is used for calling the exit function whenever an unexpected error occurs. Random is for shuffling the contents of the starting point, in order to have a random starting point. Time is used purely to get the total runtime of the algorithm. In earlier versions of the program the sleep function was also used from Time, but since has been removed.

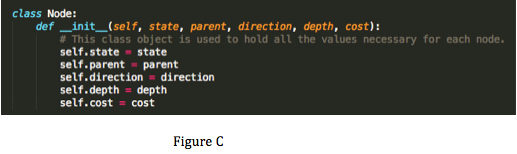


In Figure A, we get the first bit of code that checks to see what the value of \_\_name\_\_ is. Since it is \_\_main\_\_, the program now jumps to where we defined our main. In main we begin with an if-statement that checks to see that the value given in the command line exists in the argv global variable and that it is in between “0” and “4”. If it is true, the choice variable gets set to the value. If not, we go to the default value of “1” for choice. The next few lines contain feedback as to what choice was given by the user. We then step into Figure B where it contains a couple of lines of code.



In this trace, we will be using the example provide by the textbook in page 141. As such we set our start list with the values in line 278. Goal will always be set to the values in line 297. With these two values set, we inform the user to wait, as this process may take a while, and we begin out time counter. We now call the best\_first\_search algorithm function and send in the start and goal state, and choice as parameters.

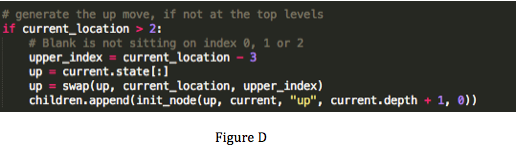
The open and closed state-lists get initialized as empty as our first action inside the function. In order to continue, we must first look at the definition of the object Node that we will be using to store each state. Figure C, shows how each Node



object will get initialized. The object receives as parameters the state array, the parent object, the direction the algorithm took to create the child, the depth of the state, and the cost to take that path. With a clearer idea of how the object gets made, we now take the list start and create a Node object with it. The values that get sent in are the state list, None for the parent, as it has None, None for direction, as no move was made, 0 for depth as it is the root, and 0 for cost since there was no decision made. The object is then appended to the open list and we prepare to enter the while-loop that will run the entire algorithm.

The first thing we notice, in line 182, is that this algorithm is meant to run until the open state is empty, or until a condition is met that breaks us out of the loop. If the open-states list becomes empty and no condition was met, the program will immediate exit out of the program. We begin by popping out the last value of the open-states list, which in this case is the start list. We set this value as current and then we check to see if current is equal to the goal state. If it did, we would append current to the closed-states list and then return that list to the main function. Unfortunately, our list is not equal to the goal state so we go into the main process of the algorithm. We need to generate the children for current, so we will call the function generate children( ) with current as a parameter and a variable named children will await for the function to return a list of children. The generate\_children( ) function begins in line 11.

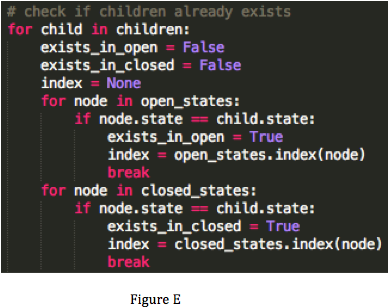
In generate\_children( ) we take the input current, and check to see what the index of the blank is. We save that index in the variable current\_location and we create a children list that will hold the children. We now check to see whether we can directions we can move in: up, down, left, and right. We have the list [2, 8, 3, 1, 6, 4, 7, 0, 5]. This means we can go up, left, and right. The if-statements will make sure to not generate the state for down, since this would cause our program go out of bounds of the list. We will further inspect, the up direction to understand what is happening.



To create a child in the up direction, we first have to make sure that the current index of the blank in not at the top of the 8-puzzle. It is not, so we can continue by making the new index of the blank by subtracting 3 from the current index. Similar numbers are subtracted or added to left, right and down. We now make a copy of object’s state variable and call it up. This prevents current from being overwritten with the next function we will do. We will now perform a swap between the value in the current index, the blank, and the upper\_index, the new index of the blank. Up gets returned as the child of current and we can now create a Node object for up. As seen in Figure D, the state that gets input is up, current is the parent, “up” is the direction, the depth is current’s depth plus one, and the cost is zero since we have not calculated it yet. We can now append this Node object to the children list. We then append, left and right’s Node object to children and return the list to children list in the best\_first\_search function. The children are as follows [2, 8, 3, 1, 0, 4, 7, 6, 5], [2, 8, 3, 1, 6, 4, 0, 7, 5] and [2, 8, 3, 1, 6, 4, 7, 5, 0].

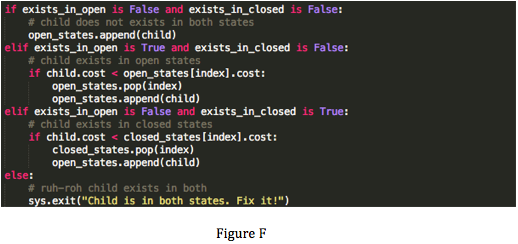
It is now time to evaluate each of the children we generated for current.

Figure E shows this process for each child and can be followed along in line 196 of the code. We begin the evaluation by, setting two variables, exists\_in\_open and exists\_in\_closed, to the value False. We then begin a for-loop that checks through the open and closed-states lists. If there is node in there that has a state that matches the child’s state, we set the respective exists\_in variable to true. We also get the index of where that Node is in the list and we then break out of the respective loop. If we were to have True for both exists\_in, our program will immediate exit out in the



following lines because we are not meant to have the same state in the both lists. We continue on in line 211, where we will calculate the cost of the child. The heuristsics will be covered later on, in this paper so for now we will only visit the calculate\_cost( ) function and then continue on with the overall algorithm.

The function calculate\_cost( ) takes three parameters. The child, which will be called current, the goal and the choice the user made at the beginning of the program. The main purpose of this function is to server as router for the incoming parameters. If the user inputs a “1” the function will send current and goal into the tiles\_out\_of\_place( ) function. If it was a “2”, the function will send the parameters to sum\_of\_distances\_out\_of\_place( ) function. If the input was “3”, the function calls sum\_of\_distances\_out\_of\_place( ) function and we get half of the cost from this function. The function tiles\_reversal( ) function then gets called and returns a value based on that part of the heuristic. If for some reason choice was another value, the program will exit immediately. Assuming that we inputted a “1”, the cost values that are returned by that heuristic for up, left, and right are 4, 6 and 6 respectively. We now return to the function that is evaluating the children generated.



For up, left, and right, Figure F shows that their exist\_in values both come back as False. As such, they get appended to the open-states list. If on the other hand, exists\_in\_open came back as True, we would have to compare the cost of child and the cost of the matching state in the open-list, thus explaining why we needed its index. If this comparison proves the child to have a smaller cost, we pop the matching state from the list and add the child in. If the child’s cost is higher, we ignore the child as there is a better existing case. If exists\_in\_closed came back as True, we would do the same thing with the matching state in the closed-states list. If the matching state’s cost was higher, we would pop it from the closed-states list and append the child to the open-states list. We are now finished, evaluating the children of current.

We have reached the last two lines of the while-loop, lines 230 and 232. Since we have finished evaluating current, we append it to the closed-states list. Since, we want to take the best path in this algorithm each time, we sort the open-states list, to give us the lowest cost value at the end of the list. We do so by calling the function in\_place\_selection\_sort( ) with open\_states as the parameters. This algorithm takes the list given to it and swaps the greatest cost value into the first index, the second greatest cost value into the second index, and so on, until all the costs are ordered from greatest to least. This is done in this way since the pop function in lists in Python remove the values from the last index. We now have (left, right, up) in the open\_states\_list, since up had the lowest cost of all three children.

We go back to the top of the list and we pop the open-states list into the current variable. Current is not the goal state so we generate the children up[2, 0 , 3, 1, 8, 4, 7, 6, 5], left[2, 8, 3, 0, 1, 4, 7, 6, 5], right[2, 8, 3, 1, 4, 0, 7, 6, 5], and down which is a copy of current’s parent. Each child gets a cost of 5, 5, and 6 respectively. Down, gets ignored since it already exists in the closed-state list, and it has a cost equal to that of the matching state. The other three get added to the open-state list and then the list gets sorted with up getting sorted to the right-end of the list. Up gets arbitrarily chosen to be there compared to left, most likely because it was the first generated into the children array. Current gets appended to the closed-states list and we begin again with the previous up now being the new current.

From this current, we get three children, with only left[0, 2, 3, 1, 8, 4, 7, 6, 5] and right[2, 3, 0, 1, 8, 4, 7, 6, 5] mattering. Left gets the cost value of 5 and right gets the cost value of 7. They both get append to the open-states list and we get that left ends up at the right-most index after the sorting. It gets popped out and becomes the new current. It has only one relevant child because it can’t go left and up, since they are out-of-bounds, and right because that was previous state. So we go down[1, 2, 3, 0, 8, 4, 7, 6, 5] with a cost of 5. It has the lowest values of the states in the open-state list, so it gets put in the right-end of the list.

We pop the open-states list to get the previous down and now it generates two relevant children right[1, 2, 3, 8, 0, 4, 7, 6, 5] and down[1, 2, 3, 7, 8, 4, 0, 6, 5] with costs of 5 and 7, respectively. The children get added to the open-states list and right gets put to the end of the list, since it has the lowest cost. We loop one last time, to have us pop the previous right, and we get a match with goal state. We add this current to the closed-states list and we now return our list to the main function.

The result variable now holds the solution path and the end variable holds the time of when the algorithm ended. By subtracting the end and begin variables we get the runtime of the algorithm in seconds. We print out our path through a for-loop, along with their depth, cost, and direction, and we print the algorithm’s runtime after the loop completes. To view the output in the terminal, go to Appendix B, following the code.

**Exploring The Heuristics:**

In the following paragraphs I will talk about the heuristics used in the program. I will use the first child generated in the trace. This state was [2, 8, 3, 1, 0, 4, 7, 6, 5] and will be referred to as current in the functions. The code begins at lines 140, 118 and 73 respectively.

The simplest of the three heuristics is the tiles out of place heuristic. For each tile that is not in the same index as in the goal state, the cost of the state increases by one. As such the calculate\_cost( ) routes the state into tiles\_out\_place( ) along with the goal state. We set sum of out of place tiles to zero, which is the variable out\_of\_place. We now take both current and goal and we zip them up. This means we can take values from each array without needing two for loops. The variables cv, the current value, and gv, the goal value, will hold the cost values of the states that are received from each array. The blank is not a tile, as such cv we ignore the case where cv is equal to zero. If cv doesn’t equal gv in that iteration, we add a one to our sum because that means the tile is out of place. We continue this until all 9 indices are checked, and then we return our cost, h(n), added with the depth of current, g(n), as a sum, f(n). In the case of current we get 3 for our h(n) and 1 for our g(n), giving us a cost, or f(n), of 4.

The second heuristic is the sum of distances out of place heuristic. This was a hard heuristic at first to implement, but with some online references, references are in the code, I was able to get the desired result. The idea of this heuristic is to get the sum of the physical distance a tile is from its goal state. To achieve this, we set a sum variable to zero again and then we initiate a for-loop from i = 1 to 9, because we do not need to worry about the blank spot. We get the index of the value i for both current and goal. We then calculate the value of each index into x and y coordinates. To get y we take the index value modulus three. To get x, we take the index divided by 3 and then we cast it to int and then take the modulus 3 of that result. With these two values for current and goal we can now use the distance formula to achieve the answer for each index. The three out of place values are 1, 2, and 8 with each value being 1, 1, and 2 tiles away from the goal tile. This gives that the heuristic value is 4, h(n), plus 1, g(n), or 5 for the cost.

The last heuristic is based on tiles that would require being in reverse order of each to get the two tiles in their respective goal positions. For example, if in a smaller scope [2,1] were the two top indices of the state and the goal’s indices had the values [1, 2]. These two could be easily be swapped and we would have them matching the goal state, but in the case of larger scope, this requires a couple of moves for that two happen. The previous heuristic runs the majority of the heuristic, but as reversed tiles begin showing up, this heuristic begins adding its values to the cost. For each index, except 0, in current we check to see if the index of the value. Using the same if-statements as when generating children, we check every direction possible for reversals. If that move is possible, we check to see if the goal state’s index is in that direction. If it is, we assign local variable called candidate the value of the current using the goal state’s index. If the candidate is equal to the value of goal using current’s index, then our reversal counter goes up by 1. At the end of checking each index, we take the reversal counter and multiply it by two. This gives us the value that we add along with h(n) value of the other heuristic to get the true h(n) for this heuristic.

**Conclusion:**

Using Python for an Artificial Intelligence application was a lot more difficult than I had imagined. I had to constantly refer to the Python documentation pages to find functions that could help me save time. A helpful function that comes into mind is zip. It allowed me to save time in calling, I think, two for-loops. I referred to their sort and sorted functions as well, but they were not able to properly sort my open-states list. That is why brought in code from Algorithms class to help me sort the list. I learned to use classes instead of the original early versions of the code where I had list of lists holding the each state and how easy they were to construct in this language. One of the hardest problems I had was realizing that “=” in Python means to “bind” instead of “equals”. This was causing data in the objects to be overwritten because every variable was passed by reference, even the copies. This was a concept that I had never seen in C or C++, where you only get values when you make a variable equal another. In the end I appreciate having a chance to make an AI application as it allowed my programming skills and knowledge in Python to grow. In the future, I plan to come back to this project and implement it in C or C++ and see how it competes to this program.