CSDS391 Intro to A.I., P1

Nicolas Slavonia, njs140

September 2022

I apologize but I definitely wrote too much. I asked Professor Lewicki about the length and he said it was ok. I worked really hard on this project and I wanted to be thorough in my explanation.

Code Design/Correctness

I decided to combine the two sections into one. I'll first explain my organization and design and then I'll go into detail and explain its accuracy. I describe everything for the 8-puzzle first and then I explain how my code is used for the 2-2-2 Rubik's cube extra credit.

8-Puzzle

The game itself is fairly simple. I have a set state method which simply sets a field called *state*. I don't bother with error checking to see if it is a valid state because I assumed we are giving it a legitimate state.

I also have a printState method which prints the current state to the interactions pane.

My move method takes a *direction* as an input. It first checks if the current state allows the given direction (I check this in the next method). It then simply swaps the correct characters in the state string to make the new state.

The possibleMove method checks to see if the given state can allow the given direction (i.e. not moving into the border). I check this by finding the *index* of the blank tile, and seeing if it can move to the up, down, left or right. For example, if the index is 0, the blank tile is in the top left corner and can't move up or to the left.

The randomizeState method picks a random number between 0 and 3 and I use if statements to allocate each number to a direction for the move method.

I then have my two search methods, one is for $A*$ search, the other is for *Beam* search. Both methods are nearly identical. They both create a new instance of their respective class used to search, ask the class to solve the state by passing it through, and then calls the getAnswers method.

What is returned from both classes in the goal node that was found from searching. I describe what the nodes do and contain in **ANode()** but basically, each node is attached to 4 children and 1 parent node. Since I return the final node, I use a while loop to iterate back up through the parents while collecting what direction each node did along the way. I then simply print the list of directions achieved. This is what the getAnswers method does.

Lastly, I have the maxNodes method and I have an additional method called visuallySolve which visually prints the answers to the interactions pane. It prints the directions along with what the state looks like along the way.

ANode

Each **ANode** represents a single state of the game. Each node can have up to four possible moves (children nodes) that follow this current state. This creates a *tree* where the number of possible nodes increase by a factor of 4. Each node also keeps track of the preceding node. Both $A*$ and *Beam* use the same nodes (i.e. same tree) to find the goal state.

Having a tree that increases exponentially (4^d, 'd' represents the depth) is a massive memory and time issue. However, we know the tree isn't that bad. Firstly, not every node can move in 4 directions, meaning not every node can have 4 children. Also, we never go back to the same state twice, meaning at most, each node can only have 3 children. This is a good start in simplifying our tree.

Each **ANode** contains the *state* of the puzzle (for this node), the *depth* of the node in the tree, the *direction* (puzzle move) the new state took, the *parent* node, the calculated heuristic *value* of the node, the four *children* nodes and the *puzzleType*. **ANode** also posses a static HashMap *allStates* which contains every *state* that has been visited.
The **ANode** class has 1 constructor, 6 methods, 1 helper method, 1 compare method (**ANode** extends Java's **Comparable** class, reasoning is explained in detail later) and 2 extra credit methods (some of the non-extra credit methods do extra credit things, I gloss over those things here and explain them in the extra credit section. Here, *puzzleType* is always equal to 0 which denotes the 8-puzzle).

ANode()

- The constructor assigns the variables for the node and creates a an array for the children.
- ∗ The **createChildren()** method assigns the new child nodes to the current node.
- The **checkVisit(String newState)** node checks if a new state (node) has been visited before, used for creating children.
- ∗ The **find1H()** method calculates its heuristic *value*.
- The **move1(String direction)** and **possibleMove(String direction, int index)** methods create new states (for the *children* nodes). This method is identical to the **Sliding-Puzzle()**'s move method but this method returns the state.
- The **compareTo(ANode node)** method is used to compare one **ANode** to another by their heuristic *value*.
- The **createHash(int hashSize)** is called once and creates a hashtable of the given size.

- The **createChildArray(int childSize)** is called in the constructor and is used to create the nodes array for the children.

ANode.createChildren()

The children of every node represent the next possible moves the given *state* can take. The move can be up, down, left or right but some moves aren't possible. Whether it is hitting a wall, returning back to the parent's *state* or revisiting a *state* that has already been seen, these nodes can't be produced.
(Pseudo code)

```
1  moves = {"up","down","left","right"}
2  for i = 0 to 4
3    newState = move1(moves(i))
4    if (newState != current state)
5      if (checkVisited(newState) == false)
6        child[i] = new ANode(newState, depth+1, direction, this)
```

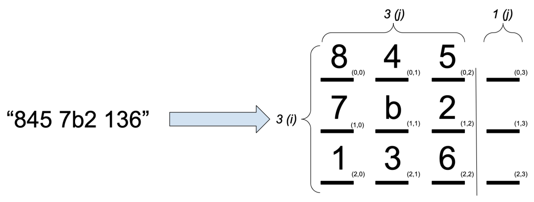
In total, **createChildren()** calls 2 other methods, **move1(String direction)** and **checkVisited(String state)**. Every node can have up to 4 children, so the **ANode** class has a variable that is an array of 4 child nodes. For each direction (line 2), make the newState (line 3) and check if the **move1(String direction)** method was able to make a new state (line 4, by a new state I mean if the state after a given move is different from the current state). If it is a new state, check to see if already exists (line 5) and if it doesn't, make a new child with that new state and direction.

ANode.find1H()

This method calculates h_1 and h_2. h_1 is an integer that counts how many game tiles are in the correct place. If a tile is not in the correct place, h_1 increments by 1. h_2 is an integer that counts how many squares (Manhattan distance) a tile is from its goal state. If a tile needs to move three places to its goal state, h_2 increments by 3. This method returns the sum of h_1 and h_2. Every state will have a sum that is larger than 0 except for the goal state. Meaning the lower the sum is, the closer this state is from the goal state.
(Pseudo code)

```
1  for i = 0 to 2
2    if (i == 1 or 2) extra++
3    for j = 0 to 2
4      index = 3i + j + extra
5      if (game tile[index] != index - extra) h1++
6      find correct indices for given game tile[index]
7        modify h2 accordingly
8  return h1 + h2
```

The game is a 3x3 puzzle board which has 9 pieces, yet the state is "845 7b2 136", 11 characters long. I decided to have nested for loops run for a total of 9 iterations (line 1 and 3) and simply pick and chose which index I wanted. The state "845 7b2 136" can be written into a 3x4 grid.



I can then use i, j and $extra$ to easily index which position I am checking (line 4) (similar to how I structured the **SlidingPuzzle.move(String direction)** method). I then check if the current tile is equal to the number it should be (line 5). In the for loops, I check what number was found at the current tile, and modify h_2. I do this by:

$$h2 = h2 + Math.abs(J - j) + Math.abs(I - i);$$

Where I and J correspond to the correct indices of the number found at game tile[index]. For example, the number 7's correct position on the board is (2,1). In the given state, 7 was found at (1,0). So the h_2 value for 7 is $|1 - 0| + |2 - 1| = 2$. So the 7 tile is two moves away from its correct position. I then return $h_1 + h_2$. (The Project 1 instructions were two compare h_1 and h_2. I do this in the experiments section. But for the algorithm itself, I found it is most efficient to return both h1 and h2, which is why the method returns the sum.)

For this example, $h1 = 8$, $h2 = 16$. So the heuristic value found by this method is 24. However, as mentioned before the final value associated to this node would be 24 + the depth of the tree that this node located at.

A* Search

The basic idea of **A*** searching is to find the most optimal path from the starting state to the goal state quickly. **A***'s algorithm does a targeted search, only looking at nodes that get you closer to the goal state, unlike **Dijkstra**'s algorithm. It does this by adding more variables to the search (**heuristics**) to understand if it is getting closer to the goal state or not.

Astar()

- The constructor makes the starting node and adds it to the list of visited nodes (*allStates*).
- The method **findShortPath()** finds the path to solve the puzzle and returns the node that equals the goal state. Although **Astar** finds the path, it is not responsible for actually solving the puzzle, it only returns the goal node.

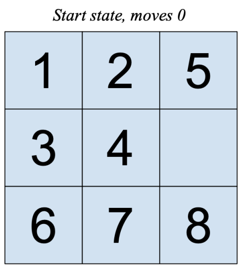
In short, the process is simple.

1. Select the starting *state*.
2. Create the *children* of the current node.
3. Calculate the heuristic *values* of the *children* nodes.
4. Add each child to a Priority Queue (based off of their heuristics).
5. Find the child with the smallest heuristic in the queue (i.e. visitList[0]).
6. Check to see if this child is the goal state.
   - If it is, return this node.
   - Otherwise, remove this child from the Array, and repeat from step 2 with this child being the current state.

(Pseudo code)

```
start.createChildren()
visitList = new PriorityQueue<ANode>()
ANode lowestN = start
Boolean found = false
while found == false
  for i = 0 to 3
    if (lowestN.children[i] exists)
      add lowestN.children[i] to visitList
  if (visitList[0].value == visitList[0].depth)
    //value = heuristics + depth, if value = depth -> heuristics = 0
    found = true
  else
    lowestN = visitList.poll()
    lowestN.createChildren()
return visitList.peek();
```

Example



Let's assume the state above ("125 34b 678") is the starting state and A* has been called. Line 1 makes the children of the starting state:

"125 34b 678" $\Rightarrow$ ["12b 345 678", "125 348 67b", "125 3b4 678", null]

The for loop in line 6 then adds every non-null child into a priority queue (line 7 ande 8):

visitList = "12b 345 678", "125 3b4 678", "125 348 67b"

We then check the first node in visitList to see if it is the goal state on line 9. It isn't, so we take the first node, and create it's children on line 13 and 14.

"12b 345 678" $\Rightarrow$ [null, null, "1b2 345 678", null]

We got back to the for loop on line 6 and add the children to the queue.

visitList = "1b2 345 678", "125 3b4 678", "125 348 67b"

Again, check to see if the first node is the goal state, if not, make children based off of that node and then add them to the queue.

"1b2 345 678" $\Rightarrow$ [null, "142 3b5 678", "b12 345 678", null]

visitList = "b12 345 678", "142 3b5 678", "125 3b4 678", "125 348 67b"

Then check if the first node is the goal state, in this case, it is! Return that node on line 15.

Beam Search

The basic idea of Beam search is to only focus on the most promising nodes. The algorithm looks at every child at a given level and keeps the k best nodes. It then only expands the previous k best nodes and looks at their children and so on. It uses the same heuristics as **A*** ($h1+h2$).

Beam(int k)

- The constructor makes the starting node and adds it to the list of visited nodes (*allStates*).
- The method **findShortPath()** finds the path to solve the puzzle and returns the node that equals the goal state. Although **Beam** finds the path, it is not responsible for actually solving the puzzle, it only returns the goal node.

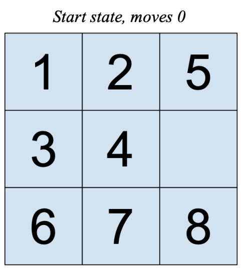
The steps for local beam search.

1. Select the starting *state*.
2. Create all possible *children* for the given tree depth and add them to an array.
3. Calculate the heuristic *values* of the *children* nodes.
4. Sort the arrays based off of the heuristics.
5. Remove the end of the array, keeping only the best k nodes.
6. Check to see if the first node, array[0], is the goal state.
   - If it is, return this node.
   - Otherwise, repeat from step 2 with the new array so we only expand those nodes.

(Pseudo code)

```
start.createChildren()
visitList = new ANode[k]
visitList[0] = start
holder = {}
index = 0
lowestN = start
Boolean found = false
while found == false
  for i = 0 to k
    if (visitList[i] exists)
      lowestN = visitList[i]
      lowestN.createChildren()
      for j = 0 to 3
        if (lowestN.children[j] exists)
          holder[index] = lowestN.children[j]
          index++
  sort holder
  if (index > k) visitList = holder[0 through k]
  else visitList = holder[0 through index]
  reset holder, index
  if (visitList[0].value == visitList[0].depth)
    found = true;
return visitList[0]
```

Example



Let’s assume the state above (”125 34b 678”) is the starting state and Beam(3) has been called. Line 1 makes the children of the starting state:

”125 34b 678” $\Rightarrow$ [”12b 345 678”, ”125 348 67b”, ”125 3b4 678”, null]

The for loop in line 9 then adds every non-null child into an Array and sort it (line 10-17):

holder = [”12b 345 678”, ”125 348 67b”, ”125 3b4 678”]

sorted holder = [”12b 345 678”, ”125 3b4 678”, ”125 348 67b”]

Since index here is only 3 (isn’t larger than k), visitList = holder. We check to see if visitList[0] is the goal state on line 21, it isn’t so we create all the children of visitList:

holder = [”1b2 345 678”, ”125 348 6b7”, ”1b5 324 678”, ”125 374 6b8”, ”125 b34 678”]

sorted holder = [”1b2 345 678”, ”1b5 324 678”, ”125 348 6b7”, 125 374 6b8”, ”125 b34 678”]

The index is larger than k, so we only keep the k best nodes (line 18). We then check if visitList[0] is the goal state, it isn’t, so do the process again.

visitList = ”1b2 345 678”, ”125 348 6b7”, ”1b5 324 678”

holder = [”142 3b5 678”, ”b12 345 678”, ”125 3b8 647”, ”125 348 b67”, ”b15 324 678”, ”15b 324 678”]

sorted holder = [”b12 345 678”, ”142 3b5 678”, ”b15 324 678”, ”15b 324 678”, ”125 348 b67”, ”125 3b8 647”]

Again, line 18 we chop off the n-k nodes.

visitList = ”b12 345 678”, ”142 3b5 678”, ”b15 324 678”

We check if visitList[0] is the goal state, in this case, it is!. Return visitList[0] on line 23.

Experiments

	8-puzzle, nodes generated						EBF (approx)	
b	**A*($h1$)**	**A*($h2$)**	**A*($h1$+$h2$)**	**beam(1000)**	**beam(100)**	**beam(10)**	**A*($h1$+$h2$)**	**beam(100)**
2	6	6	6	9	10	10	2.363	3.010
4	11	10	10	32	32	32	1.761	2.369
6	17	15	15	102	99	67	1.546	2.152
8	32	22	22	283	282	101	1.457	2.021
10	70	33	37	751	606	137	1.405	1.895
12	159	54	59	1,996	940	169	1.394	1.768
14	350	102	100	4,843	1,271	204	1.375	1.664
16	809	188	171	8,007	1,598	-	1.360	1.584
18	1,987	325	305	11,175	1,927	-	1.365	1.521
20	5,019	632	457	-	2,256	-	1.351	1.470

These results are gained from averaging 100 random puzzle states. I left my test method in comments at the bottom of each class (you may not get the same answers as above because I changed the random number seed).

a)

Percentage of solvable puzzles						
max	**A*($h1$)**	**A*($h2$)**	**A*($h1$+$h2$)**	**beam(1000)**	**beam(100)**	**beam(10)**
500	2.6	23.1	30.7	0.2	0.2	40.1
1,000	6.4	45.2	51.9	0.3	1.3	93
2,000	14.4	69.3	80.5	1.4	13.6	99.9
5,000	27.3	92.1	95.6	2.8	99.6	100
10,000	37.7	98.0	99.8	10.8	100	100

For the results above, I counted the number of times the searching method was able to solve the random state out of 1,000 random states.

b) H2 is a better heuristic than h1. The Manhattan distance provides a better understanding of how many moves we are from solving the puzzle. However I found that combining both heuristics is actually best.

c) A* with h1 begins to increase into the thousands of nodes when the depth is around 16. Large values of k for Beam are widely unhelpful while very low values prevent you from finding a solution. Using Beam(100) provides a search that is better than A* with h1, but not as good as A* with h2 or h1 + h2. These methods only every reach a couple hundred nodes.

d) I was unable to find a state that wasn't solvable. The randomizeState() method only produces solvable puzzles so given a large enough max nodes limit, they were all solvable.

2x2x2 Rubik's cube, nodes generated			**EBF** (approx)	
b	**A*($h1$+$h2$)**	**beam(100)**	**A*($h1$+$h2$)**	**beam(100)**
2	24	127	4.898	11.269
4	200	1,878	3.429	6.582
6	1,871	3,742	3.072	3.941
8	20,739	5,657	3.076	2.942
10	200,108	7,552	3.110	2.442
12	1,089,599	9,468	3.105	2.143
14	-	11,318	-	1.947

For the Rubik's cube, local beam search is much more effective. It takes a couple thousand nodes to solve the given cube in less than 14 moves where as A* can take millions of nodes. The effective branching factors also show how much the A* tree grows compared to the beam tree.

I should mention something important regarding the tests I conducted. It applies to both puzzles but is extremely apparent for the Rubik's cube. If you shuffled a cube and and solved it using Beam(100), there is a high chance it was solved in 50ish moves while checking 200,000 nodes. Beam Search does not guarantee optimality. The tests above only take into account when beam(X) solved it in the given depth. Taking this into account, the conclusions don't change, A* with h2 or h1 + h2 are best for the 8-puzzle and beam search is best for the cube. Fun fact, the lower the k is for beam search, the narrower the tree is. Shuffling the cube, you can use Beam(1) and get effective branching factors like 1.001 (may take a couple thousand moves to solve though). I used beam 100 above to show the difference in node count for solving the Rubiks2 cube. If you want to solve it optimally, I recommend using beam(1000) or beam(2000).

Discussion

a)

A* with good heuristics is by far the best searching algorithm for the 8-puzzle. Using h2 or h1 + h2 produces a low amount of nodes as well as a very low branching factor. The minimizes the space needed to solve the puzzle as well as minimizes the number of comparisons needed which uses less time. With better heuristics, A* may even be able to generate less nodes to solve the puzzle. However, looking at the percentage of solvable puzzles, a good argument can be made for the reliability of Beam search.

The 2x2x2 Rubik's cube has the opposite effect. This makes sense. A cube can be very close (1 or 2 pieces off) to being solved. Yet it may require a full shuffle of the cube in order to solve it. Beam search will take the k best shuffles and expand them together. A* will linger around these 'close' states before actually solving it. This is because the shuffled states will appear near the end of the priority queue. A* may find shorter solutions, but Beam search is quicker and searches less nodes.

b)

The main difficulties for creating the 8-puzzle were the searching algorithms and the heuristics. Making the game itself had it's own minor inconveniences. For the heuristics, the Manhattan distance was a little difficult to implement. I wrote down a game board on a piece of paper and worked out a way I can calculate h2 for every piece on the board. Once I had an idea of what to do, coding it wasn't terribly difficult. Implementing A* and Beam weren't that hard. I knew we needed some sort of a queue and a while loop. Since we learned both algorithms in class, coding in the details was easy. I made some mistakes along the way such as having an infinite while loop, incorrect sorting methods, adding incorrect nodes to the queue and more. It didn't take long until they were both up and running. Since the ANode class really does most of the work, the algorithms themselves were fairly straight forward. Fun fact, I originally made the ANode class to be only used by A*, hence the 'A' in the name. I then realized Beam search uses the exact same tree and nodes so I let Beam use it. I never changed the ANode class name because I already have a 'Node' class from CSDS 132. Overall, this project wasn't that difficult. For the 8-puzzle, I didn't have any moments where I was totally stuck. I just solved one small problem at a time and by the end it worked.

Testing the program has some issues. Every time I made a new method, I'd make sure it works. When I got the move method to not return any errors, I tested it a lot. I would know if the method worked by simply looking at the state. Same for the heuristics, I'd give it a state and have it find h1 and h2. I could then do it myself and see if it was correct. I'd double check every child would be made and check if they were entered into the HashTable in the ANode class. With every method, I could manually see if it was correct or not. When it came to testing the searching algorithms, I kinda just had to trust it. I would start with easy states (2-3 moves away from being solved) and it would work. When it came to solving a state that needed 25ish moves, I can't really know if the algorithm is perfect. I can't tell if the solution it gave me is the most optimal solution. I'd make sure I was getting a valid solution, but knowing if it skipped a solution or was checking too many nodes is hard to know. These issues with testing apply to the Rubik's cube as well.

The Rubik's cube had three extraordinarily frustrating issues. The first was creating the cube and all of its moves, the second were the heuristics and the third was the HashTable. Making the Rubik's cube class and simply being able to type move("R'") or move("D") was extremely frustrating and easily the hardest part. Every move changes 12 pieces. For every move, I needed to figure out which 12 characters in the state were going to move, and where they'd go. Every time I'd think I got it, I'd test it, I'd then find 1 piece rotated incorrectly making the cube unsolvable. The heuristics weren't easy either. It wasn't so much that they were hard, it was more that I didn't know what to do. I looked up the Ortega method and tried coding 3 sets of heuristics based off of that. The idea was I'd have the cube get close, then closer and closer, then solve it. Turns out the best way to go about it is simply solving the cube immediately (i.e. the only heuristic is 6 opposing sides). I must confess, I have not been able to solve the last issue. Take the state "WWWW BBBB RRRR GGGG OOOO YYYY". I hash the string using Java's built in hash code and then put that into the HashTable. Take the state "WWWW RRRR GGGG OOOO BBBB YYYY". This is the exact same state as before, but the entire cube has been rotated to the left. However, the hash code is different, so my HashTable sees this as a new state, i.e. a new possible child. I now have the same state in HashTable twice. Each state can have 24 different orientations, meaning worst case scenario, each state can be in the HashTable 24 times I have tried adding each new state along with its 23 other rotated states to the HashTable but this process is time consuming. This also produces a set of new problems. Regardless, I am not really sure how to fix this so I have removed this 'fix' and A* and Beam search can still solve the cube, but they will revisit states. At most, A* takes 5 seconds, Beam(1000) is nearly instantaneous.

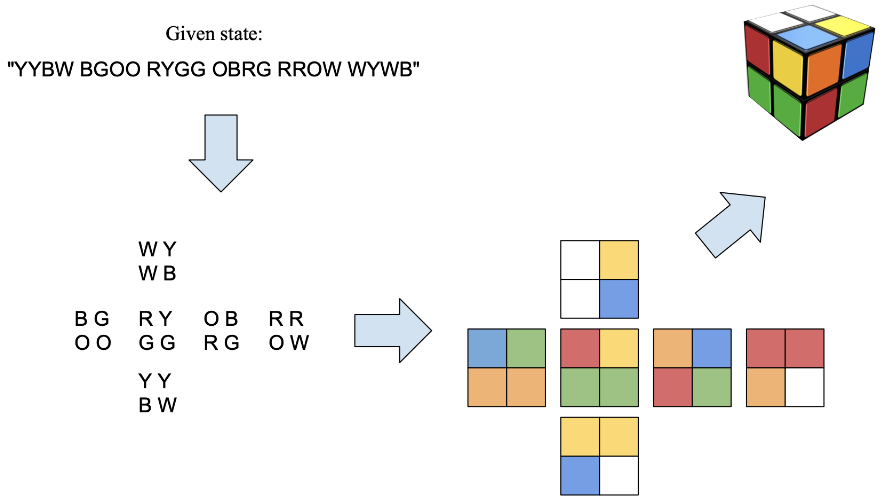
Extra Credit - 2x2x2 Rubik's Cube

To avoid repetition I will only explain what needed to change between the 8-puzzle and the 2x2x2 Rubik's cube. In short, I didn't change anything besides the obvious. The structure of the problem is the same. You have a puzzle, you can make moves, we are tasked with finding the solution. The rubik's cube has a **Rubiks2** class that is new and uses the same **ANode**, **Astar** and **Beam** classes I mentioned above.

2x2x2 Rubik's Cube

The Rubik's cube class has the same structure as the 8-puzzle class. It has a set state method, move method, A* and Beam search method, a randomize state method, a max nodes method and a visual solver. Again, to avoid repetition, I won't spare you the details. It is the same thing except the game moves are different.

A state is represented as "WWWW BBBB RRRR GGGG OOOO YYYY" where the W side is the bottom, R is the side facing you, B is to the left, G is to the right, O is the back and Y is the top. Each letter represents a piece on that side. Looking at "RRRR", the side facing towards you, the first R is top left corner piece, the second is the top right corner piece, the third is the bottom left corner piece and the fourth is the bottom right corner piece. Each side has the same orientation. For example, some state would look like:



ANode changes

Again, the structure is the same but things do change. Firstly, there are different moves (different puzzle) and the heuristics are different. The create children method uses an if statement to determine which moves to conduct.

The heuristics are different. My current set up is to simply solve it. What I mean is, in the solved cube, white opposes yellow, blue opposes green and red opposes orange. The higher the count of pieces correctly opposing their pair, the closer we are to solving the cube. We also need to take into account the fact that each side needs to be the same color. If each side posses a single color and each side opposes their correct pair of colors, the cube has been solved.

(Pseudo code)

```
h1,h2 = 12
pairs = "WY, BG, RO"
index1 = {0,5,10}
index2 = {25,15,20}
for i = 0 to 3
  for j = 0 to 6 (j += 2)
    for k = 0 to 4
        if ((state.charAt(index1[i] + k) == pairs.charAt(j)) &&
            (state.charAt(index2[i] + k) == pairs.charAt(j+1))) h1--
        if ((state.charAt(index1[i] + k) == pairs.charAt(j+1)) &&
            (state.charAt(index2[i] + k) == pairs.charAt(j))) h2--
return min(h1,h2);
```

The 'i' loop runs for each pair, the 'j' loop runs for every other side and the 'k' loop runs for each piece. The if statements check if the side we are on (index1) has color pairs(j) and compares that to the opposing side (index2) with color pairs(j+1). There are two if statements, this is important because we could be looking at a white piece opposing a yellow piece, or a yellow opposing a white. Relative to the string (state) these are different events.

A* and Beam search changes

The only difference for both methods is what goes on in the constructor. In the 8-puzzle, I make a HashTable with the size of 10,000. For the Rubik's cube, the size is 1,000,000. This is because the Rubik's cube will visit more states than the 8-puzzle. That is the only difference for the searching algorithms.