* A description of your program, including a separate description of the the data structures used in the search algorithm, and a separate description of the heuristic you used;

Serialize\_Cube()

This function takes an instance of the cube class, and serializes the arrays which describe its state into a single tuple, which it then returns.

Unserialize\_Cube()

This function takes a tuple which represents the cube state, and constructs and returns a Cube instance using portions of the length of the tuple. This is used when a node is visited.

Class Node

This node is its own custom class of object. It is given a cube state and parent node when initiated, if no parent is provided, the parent is None.

It stores the cube state as a tuple, as well as the move made to reach it. It then unserializes the cube tuple and gets its h value by calling the Moves\_To\_Solved\_Heuristic() Cube class function.

Also, it checks if its parent value is None. If it isn’t, it increments its g-value (depth in the tree).

Otherwise, it sets it’s depth to 0.

It then sets its own f value based on the sum of f and g, and declares a vector of child nodes, but does not populate it.

expand\_node()

Takes a node as a parameter and unserializes the cube state of the node 6 times, performing a unique leftward movement to attempt to move it closer to the solved state. It then serializes the cube after the move was performed and places it into a node structure, and updates the move\_made value of that node with the move performed. It does this for each of the 6 possible counter-clockwise moves. When all child nodes are generated, it then populates the passed node with the child nodes in its vector of child nodes.

Because my randomizer in my cube class has been altered to only perform right turns in the scramble, I only make 6 children, making left turns to solve.

trace\_path()

takes a node as a parameter, and traverses up the chain using the node.parent attributes, and also populating a list of the node.move\_made attributes of each node it visits. It then reverses this list to print out the steps in the correct order for altering the cube to find a solved state, trace path doesn’t print the steps, it returns the reversed list.

Star\_Search()

Takes the root node as a parameter.

Set() used to store visited\_nodes to store nodes already visited for comparison before expanding a node, as suggested by Dr. Harrison.

The frontier is a min heap, organized by the minimum f-value.

A counter integer is used to ensure tie breaks using the first-generated node, so that later, nodes are never compared.

It then pushes the root node onto the heap before beginning a while loop, which runs as long as the frontier is not empty, or until a break condition is met.

While loop:

Inside the while loop, the min of the heap is popped, and inspected. It checks if the cube state is in the set of visited nodes, and re-starts the current while loop if it is. If it is not in the set of visited nodes, the loop then adds it to the visited nodes, and checks

if it is a solved state by seeing if its h attribute is zero. If it is, it prints the list generated by trace\_path, and then returns the length of the frontier.

If it is not a solved state, it expands the node and inspects each child to see if it has been seen already, if not, then it adds the child to the minheap. Counter is incremented for each child inspected, to ensure a unique value for each node, with minimum value meaning earliest inspected.

main()

First, my main function initiates k with the value of 3, before entering a while loop with the condition “while k <= 20”. For each k, 5 k-scrambled cube instances are created, serialized, and then passed as root nodes to the Star\_Search function. The star search function returns the size of the minheap when a solved state is found, but it also prints out the moves-to-solve the cube. The main function sums the returns of the 5 calls to Star\_Search() and divides them by 5, finding the average size of heap for the current k-value, and printing the current k-value, and average.

Data structures used in search algorithm:

My previously used face and cube classes were used, as well as a serialized tuple to compress the cube state. This was done using serialize and unserialize helper functions which combine the lists from the cube class into a tuple, and separate the tuple into faces of a cube respectively.

I defined my own node structure, which contains the cube state as a tuple, a pointer to the parent, a vector of child nodes, a vector to store two strings for backtracing the moves made on the cube, a self.h value, which first unserializes the cube state, and then runs the heuristic on the cube, a self.g value, which is 0 if there is no parent, and parent.g + 1 if there is a parent. The self.f value is the sum of self.h and self.g.

I also used a set() for the closed list, to check if a node was seen before, and for the frontier, a heapq was used, with the f value as the first minimum value to be used, and for tie breaking, the g value, and for tie breaking for this, a counter (integer). This was so I would not have to write a custom node comparison function for my node class. Notably, my cube class is where the heuristic code can be found.

A custom node class data structure was used, which takes the serialized cube state and unserializes it to set its own h, g, and f values when initiated as a node.

Describe the heuristic: I have not changed my heuristic except to remove a redundant attribute update. So I have copied and pasted the description of my heuristic from the last report, which is my own original heuristic, and received full credit in the first project.

* This heuristic calls a helper function on each of the 6 faces of the cube, and places the count of non-distinct "stickers" on each face into an array in the order…
  + [top, left, front, right, back, bottom] # array of repeats
* I find the number of faces that have more than 0 repeated non-distinct "stickers"
* if the max occurs on more than one face, and all of the faces of the cube have non-distinct "stickers", and the max is 4 or greater
* **Assumption:**
  + There is more than one solve state, therefore the heuristic guesses the minimum number of moves it would take to solve. Not the number of moves it would take to return to the original solve state.
* **Justification:**
  + This heuristic is **admissible** because it never overestimates the number of moves required to solve the cube. The estimate is based on the maximum number of repeated stickers on a single face, divided by 3; since a single move can fix at most 3 stickers on that face, I divide by three and take the ceiling of that. By rounding up, it ensures it does not undercount the minimum number of moves needed for that face. In the case that the max is >= 4, and there is more than one face that is that scrambled, it is impossible for a single move to resolve them simultaneously, the heuristic adds +1 to account for the extra move that must be made, since you could theoretically fix 2 faces with two moves, but you would not be able to fix all six faces of the cube in two moves, hence my comparison and justification for a +1 case if all those conditions are met. Therefore, the heuristic provides a safe lower bound on the true cost to reach the goal state.if these criteria are not the case, I simply return the ceiling of the max / 3 without the + 1. And since the lower bound relies on each move being very ideal, it will never overestimate the number of moves until solved.
  + I have run this more than 30 times. It never overestimates, or declares it to be 0 unless the cube is solved (move estimate is 0).
  + move estimate is never zero unless the max in the array is 0.
* Instructions on how to compile/run your code;
  + You have already unzipped the file
  + The file is called A-Star-Cube-Project
  + Inside this file you will find the three .py files used, as well as the project file.
  + Launch VS Code, and open the A-Star-Cube-Project folder.
  + Ensure you have python3 installed
  + You can run it in VS Code using the run program button in the top right corner.
  + You may be prompted for whether you trust the author, respond as needed to run the code (just this once, or always)
  + This will output to the terminal in VS Code, and terminate after you press enter. This program has been known to eat into SSD space for some unknown reason when K > 10. This is not permanent and goes away when the program is terminated.
  + You can also run the code by double clicking main.py, assuming you have python3 installed. Or right-click main.py and select “run with” if python is not your default method.
* A plot of the average number of nodes expanded in the last iteration of A\* as a function of the actual distance to a solution (the parameter k);

**LLM used: ChatGPT 5 Instant, below are the prompts and responses:**

Prompt 1:  
frontier = PriorityQueue() # put root node into frontier queue

frontier.push(root\_node.f, root\_node) # while loop to a\* search

while frontier.size() != 0:

present\_node = frontier.pop()

How do I pop the node with the lowest f value?  
  
**Chat GPT response:**  
Good question — this is the heart of A\*.

In Python you usually don’t write your own PriorityQueue; instead you use the built-in **heapq** (min-heap). With heapq, the *smallest* item is always popped first.

**✅ Using heapq**

import heapq

frontier = [] # this will be our heap

# push root (priority = f value)

heapq.heappush(frontier, (root\_node.f, root\_node))

while frontier:

# pop the node with the lowest f

f\_val, present\_node = heapq.heappop(frontier)

# check if goal

if present\_node.h == 0: # or some is\_goal(present\_node)

return trace\_path(present\_node)

# expand children

for child in visit\_node(present\_node):

heapq.heappush(frontier, (child.f, child))

**🔎 Why this works**

* Each entry in the heap is a tuple (priority, node).
* Python compares tuples by the first element → so it uses f as the priority.
* heappop always returns the tuple with the **lowest f**.

**⚠️ If you really want to use queue.PriorityQueue**

You can, but it’s slower. You’d do the same:

from queue import PriorityQueue

frontier = PriorityQueue()

frontier.put((root\_node.f, root\_node))

while not frontier.empty():

f\_val, present\_node = frontier.get()

...

✅ Best practice for A\*: **use heapq with (f, node) tuples**.  
That guarantees present\_node is always the one with the lowest f.

Do you want me to also show you how to handle **ties** (when two nodes have the same f value), so the heap doesn’t complain about comparing Node objects directly?

(I did not ask it to do so)

**Prompt 2:**

# make an A\* recursive function using priority queue and hashtable

def Star\_Search(root\_node: Node):

# make the list for tracking states already visited.

# only use the serialized value.

visited\_nodes = set()

# See citations in word document, the following is based on a repsonse given by chat GPT after being supplied my code using a priority queue.

# minheap

frontier = []

# put root node into frontier heap

heapq.heappush(frontier, (root\_node.f, root\_node))

# while loop to a\* search

while len(frontier) != 0:

# pop lowest f node

f\_value, present\_node = heapq.heappop(frontier)

# check if node has been visited before

if present\_node.cube\_state in visited\_nodes:

# get new node from heap if it has been visited before

continue

# Add the visited node to the list of visited node

visited\_nodes.add(present\_node.cube\_state)

# check if present node is solved state

if present\_node.h == 0:

# print the path

print(trace\_path(present\_node))

# return the size of frontier

return len(frontier)

else:

# expand present node

expand\_node(present\_node)

for child\_node in present\_node.children:

# put the children into the frontier

heapq.heappush(frontier, (child\_node.f, child\_node))

# if while loop terminates without returning, there was an error

print('error, while loop terminated without finding solution')

return -1

**Here is my current code, are there any fixes to be made?**

**ChatGPT**

Perfect — thanks for pasting the whole Star\_Search. I’ll mark the **fixes you need** directly in your function so it’s concrete. The main problems are:

1. **Tie-breaking in the heap** → you need a counter or a (f,g,...) tuple, otherwise Python may crash if two nodes have the same f.
2. **Consistency in visited** → best practice is to add to visited\_nodes right after popping, before expansion.
3. **Comments / cleanup** → simplify continue logic and make sure everything is clear.

**✅ Fixed version**

def Star\_Search(root\_node: Node):

visited\_nodes = set() # track visited states

frontier = []

counter = 0 # unique counter for tie-breaking

# push root node into frontier heap

heapq.heappush(frontier, (root\_node.f, root\_node.g, counter, root\_node))

counter += 1

while frontier:

# pop lowest f node

f\_value, g\_value, \_, present\_node = heapq.heappop(frontier)

# skip if already visited

if present\_node.cube\_state in visited\_nodes:

continue

# mark visited as soon as it’s popped

visited\_nodes.add(present\_node.cube\_state)

# goal check

if present\_node.h == 0:

print("Solution path:", trace\_path(present\_node))

return len(frontier)

# expand children

expand\_node(present\_node)

for child\_node in present\_node.children:

if child\_node.cube\_state not in visited\_nodes:

heapq.heappush(frontier, (child\_node.f, child\_node.g, counter, child\_node))

counter += 1

print("Error: search terminated without finding solution")

return -1

**🔑 Changes made**

* **Heap entries** are now (f, g, counter, node):
  + f = A\* priority
  + g = depth (secondary tie-breaker)
  + counter = unique id to guarantee no crashes
  + node = the actual object
* **Pop** with:
* f\_value, g\_value, \_, present\_node = heapq.heappop(frontier)
* **Visited marking** happens **immediately after popping**, not after expansion. That ensures you never expand the same state twice.
* **Push** children with (child\_node.f, child\_node.g, counter, child\_node).

**⚡ Why these changes matter**

* Without the counter, if two nodes have the same f (and g), Python will try to compare Node objects → TypeError.
* Marking nodes visited after popping is the standard A\* pattern: a node is “expanded” once, when it comes off the heap with the lowest possible cost.

👉 Do you want me to also show how to make trace\_path print **just the move sequence** (like [(top,left), (front,left)...]) instead of nested lists of strings, so your output is cleaner?

My Final version of A\* function after implementing the changed suggested by ChatGPT, the format may be odd:

# make an A\* recursive function using priority queue and hashtable

def Star\_Search(root\_node: Node):

    # make the list for tracking states already visited.

    # only use the serialized value.

    visited\_nodes = set()

    # See citations in word document, the following is based on a repsonse given by chat GPT after being supplied my code using a priority queue.

    # minheap

    frontier = []

    # counter for tie breaking

    counter = 0

    # put root node into frontier heap

    # g is compared second to break ties, and then if needed, uses counter so it never tries to compare nodes.

    heapq.heappush(frontier, (root\_node.f, root\_node.g, counter, root\_node))

    # use of counter was suggested by chatGPT

    counter+=1

    # while loop to a\* search

    while len(frontier) != 0:

        # pop lowest f node

        f\_value, g\_value, count, present\_node = heapq.heappop(frontier)

        # check if node has been visited before

        if present\_node.cube\_state in visited\_nodes:

            # get new node from heap if it has been visited before

            continue

        # Add the visited node to the list of visited node

        visited\_nodes.add(present\_node.cube\_state)

        # check if present node is solved state

        if present\_node.h == 0:

            # print the path

            print('Solved, here is the path', trace\_path(present\_node))

            print('here is the cube')

            the\_solved\_cube = Unserialize\_Cube(present\_node.cube\_state)

            the\_solved\_cube.Print\_Cube()

            # return the size of frontier

            return len(frontier)

        else:

            # expand present node

            expand\_node(present\_node)

            for child\_node in present\_node.children:

                # make sure child is not in visited nodes

                if child\_node.cube\_state not in visited\_nodes:

                    # put the children into the frontier

                    heapq.heappush(frontier, (child\_node.f, child\_node.g, counter, child\_node))

                    # increment counter used for tie breaking to open the node made first

                    counter += 1

    # if while loop terminates without returning, there was an error

    print('error, while loop terminated without finding solution')

    return -1