CS-463G-001: Program #2, A* on the Master Pyraminx

INCLUDED FILES AND DIRECTORIES:

\Code

master_pyraminx.py	tester.py
solver.py	randomizer.py

\Experiments *Documentation output of the tester code will go here.*

\Demo Experiments *Unformatted experiment output. (Examples)*

‘Exp (k=0) – 160064….csv’, etc.

\Formatted Results *Review these for formatted/analyzed data.*

‘Demo_Exp (k=0).xlsx’, etc.

\Documentation

structure_documentation.txt

HOW TO RUN:

Running the Solver:

- Start with by running the program from the command line.

  ```
  py .\solver.
  ```

- After starting the program, press any key to continue.
- Then, input the *integer* number of random rotations to be made (0-20) I recommend no more than 6 to guarantee an expedient result.
- Wait for the solver to return a solution to the terminal. If the program is taking too long to your liking, press Ctrl+C to quit, then start over from Step 1. Otherwise, once a solution is returned, steps 2 through 4.

Running the Tester:

The testing program should be run from the command line using the following format and arguments:

```
py .\tester.py [n_moves] [n_tests] [Document? y/n]
```

[n_moves]: number of random rotations (0-20, l.t. 7 recommended)

[n_tests]: number of test-runs for this experiment

[Document? y/n]: character input indicating if output is saved

Should a test-run fail for any reason (i.e.: KeyboardInterupt, time exceeded), a failure will be recorded to any documentation. The user should then input if experiment should be shut down entirely (‘y’/‘n’). [For massive experiments, this capability can be commented out.]

PROGRAM WRITE-UP:

Pyraminx Data Structures: Please review previous README file (structure_documentation.txt) for general info on the pyraminx class.

Some changes have been made to the class since the previous submission:

- A .move(n) method was implemented: “Given an integer – or set of integers – rotate the pyraminx in the given order of steps.” Allows for easy verification of solutions, cuts down on code required for the .randomize(n) method.
- Additional comments, function descriptions have been added to methods directly called by the driver.
- Code-blocks for bit rotations have each been reduced to a single line of code (previously 4 lines each).
- Program can now accommodate set requests for randomization, i.e.: only use clockwise (ODD) or counterclockwise (EVEN) moves to randomly rotate the pyraminx.

Solver Data Structures:

class Node:

__init__(self, pyraminx, parent=None, next_move=None):

Given a state of the Master Pyraminx, records bit-pattern, evaluates heuristic, and maintains list of moves from initial state.

Class Object’s Public Attributes:

self.parent: Pointer to the Node’s parent, if one exists. (If first node, parent=None)

self.bits: Record of the bit-pattern of the given pyraminx.

self.moves: List of moves taken from the initial state. (If first node, moves=[])

self.g: number of integer steps taken from initial state. (If first node, g=0)

self.h: The recorded, float heuristic value of the given bit-pattern. (h always equals heuristic(self))

self.f: The evaluated, float sum of the heuristic and steps taken for the Node’s current state. Generally, lower, the better!

solve(p_start, set=None, testing=False):

Given an initial state of the Master Pyraminx, uses A* to return an solution set in approximately 10 minutes or less.

Returns a list of moves from the initial state to the solution, the runtime in float form, and the number of nodes searched (or expanded) by the solver in integer.

Use set=ODD or set=EVEN to restrict solver to clockwise, counter-clockwise moves. Use testing=True to ensure program will not end if solver fails.

Uses several data structures, methods local to the function:

- Lists of searched and to-be-searched nodes. (closed_list, open_list)
- For each given node, solver randomly selects valid numbered moves from an array (move_nums) to fairly order, evaluate and append the node's children in the open_list – should their heuristic have the same value.
- Solvers uses the lists.sort() and attribute get methods (attrgetter()) to sort the open_list of nodes by their heuristic evaluation: f.
- Note: Node initialization automatically calculates and evaluates the heuristic of each new Node. So, heuristic() is never called by the solve() function.

heuristic(p_state):

Given state of the master pyraminx, returns float guesstimate of steps toward a solution. Developed largely with Tony Chiu.

Expanded upon in the *Heuristic Description* section.

Main Driver Code:

- Initializes a solved-state Master Pyraminx (p).
- Then – per the user input – program randomly rotates the pyraminx some k times, where k is between 0 and 20.
- Program calls solve() to attempt a solution to the pyraminx in under 10 minutes. (Reliable for up to k=6)
  - If a solution is found, it is returned, implemented through the pyraminx's move method, and the user is given an option to continue requesting random states and solutions. They may also end the program using Ctrl+C.

- If a solution is not found – due to either keyboard interrupt or the problem timing out – the program quits with an error code 01.

Tester Data Structures:

experiment(n_moves, n_tests, documentation):

Runs and documents randomized runs of the Master Pyraminx solver where requested variables k=n_moves and n=n_tests are given.

Boolean flag (documentation) used to create and maintain record of the experiment. Flagged True/False on user request.

Successful and failed (i.e.: interrupted or timed-out) runs can both be documented by the function by opening and writing to a CSV file, which holds the Solution Length, Number of Nodes Expanded, Runtime, and Solution Move-Set of each test-run.

Should a test-run fail for any reason, the user will be asked to confirm if the experiment should continue – answering ‘y’ or ‘n’. [For massive experiments, this capability can be commented out.]

Heuristic Description:

Credit for this heuristic design largely goes to Tony Chiu.

Each face of the pyraminx has 7 centers:

1 face center – in position 6(G);

3 edge-center bits – in positions 5(F), 7(H), 12(M); and

3 non-edge-center bits – in positions 2(B), 10(K), 14(O).

All bits in on a Master Pyraminx face can be broken up into four contiguous sets of four bits each – called triangles:

a “top" triangle, centered at bit 2(B);

a “left” triangle, centered at bit 10(K);

a “center” triangle, centered at bit 6(G);

and a “right” triangle, centered at bit 14(O).

That said, let us define our heuristic as a factored sum of two values:

One, the number of bits in each triangle that need to be colored the same as their triangle’s center bit; and

Two, the number of non-edge-center bits (1-3) that need to be colored the same as the face’s center (6/G).

The factor of these values – one-half and one-fourth, respectively – was found through experimentation.

Furthermore, it was found that the sum of the square roots of these factors acted as a more accurate heuristic – decreasing the runtime and number of nodes expanded in early test-runs.

So, h can be best defined as $(.5a)^{0.5} + (.25b)^{0.5}$, where

a = the number of bits off-colored from their respective triangle’s center bit – on each face; and

b = the number of triangle, non-edge center bits off-colored from their face’s center bit (6G) – on each face.

Heuristic Pseudocode:

```
Consider a pyraminx state with a given bit-pattern (B):

    if B equals the bit-pattern of a solve-state pyraminx:

        Solve-state is found – return 0

    Initialize two separate counters to be used in the
    proceeding calculations.

    For each face, define 4 smaller triangles of 4 bits each:

        the “top" triangle, centered at bit 2(B),
        the “left” triangle, centered at bit 10(K),
        the “center” triangle, centered at bit 6(G), and
        the “right” triangle, centered at bit 14(O).

    For each face of the pyraminx:

        And for each “triangle” on the face:

            Consider the number of bits which are off-color
            in comparison to the triangle’s center bit.

            For each bit in a triangle off-color from its
            respective center bit,

                Iterate the first counter by a half-step.

        Consider the number of center-bits (B,K,O) which are
        off-color in comparison to the face’s center bit (G).

        For each triangle, non-edge center bit off-color from
        face’s center bit (G),

            Iterate the second counter by a fourth step.

    Return the sum of the square roots of both counters.
```

EXPERIMENT, RESULTS AND DATA ANALYSIS:

The experiments discussed here can be reviewed in the Formatted Results subdirectory, for steps (k=0) through (k=7).

In each experiment, 500 test-runs were attempted in under 10 minutes - 600 seconds – for a specific value of k (between 0 and 7). If a solution could be achieved with less than k steps of output, its results were deemed unnecessary to the problem. (This is because – as more steps are used to randomly generate states – some steps can cancel each other out. This becomes increasingly true as k grows.)

A summary of the results can be seen on the next two pages.

For (k=0):

Average Solution Length	Average Nodes Expanded	Average Runtime
0	1	0.001563762
Total/Valid Experiments		
500		

For (k=1):

Average Solution Length	Average Nodes Expanded	Average Runtime
1	2	0.014532467
Total/Valid Experiments		
500		

For (k=2):

Average Solution Length	Average Nodes Expanded	Average Runtime
2	3.286	0.028176426
Total/Valid Experiments		
500		

For (k=3):

Average Solution Length	Average Nodes Expanded	Average Runtime
3	5.777108434	0.048725826
Valid Experiments		
498		

For (k=4):

Average Solution Length	Average Nodes Expanded	Average Runtime
4	14.88548057	0.124739897
Valid Experiments		
489		

For (k=5):

Average Solution Length	Average Nodes Expanded	Average Runtime
5	53.46680498	0.560348684
Valid Experiments		
482		

For (k=6): First instance of an aborted problem:

(~3500 nodes expanded in ~550s)

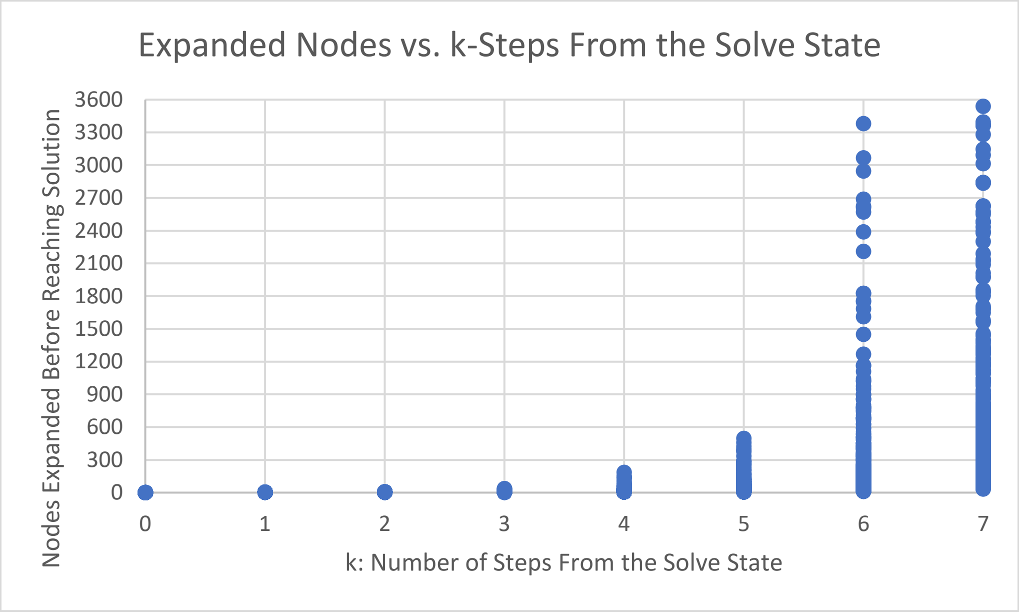
Average Solution Length	Average Nodes Expanded	Average Runtime
6	269.2653509	16.25692762
Valid Experiments		
456		

For (k=7): Forty instances of aborted problems – for this reason, I do not consider (k=7) to a complete test-run. But it took 12 hours on six-year old hardware, so it is what it is.

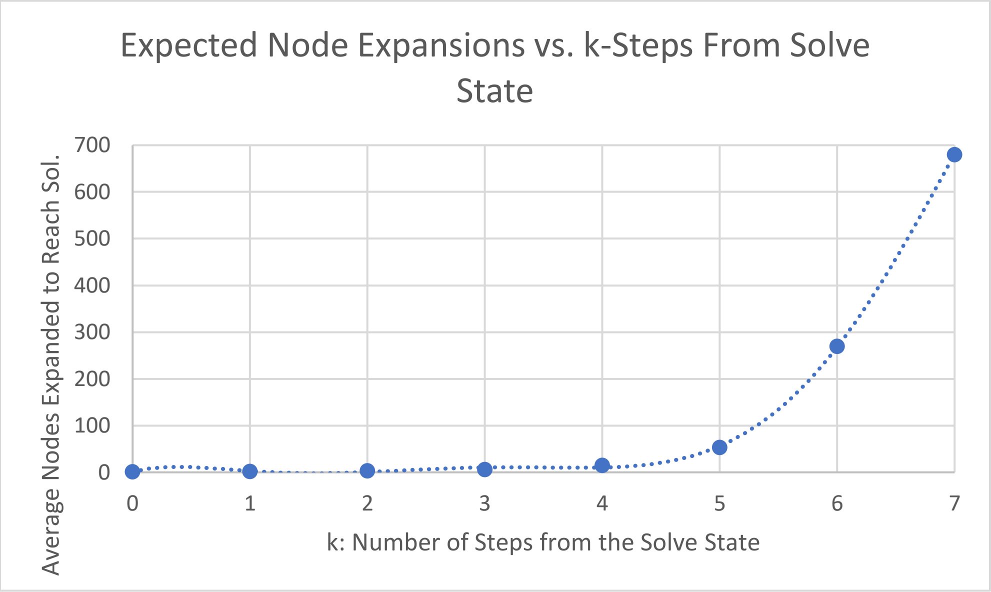
Average Solution Length	Average Nodes Expanded	Average Runtime
7	679.9537275	47.71438876
Valid Experiments		
389		

Analysis on the following pages.

Although the heuristic solver works well for (k <= 6), it is clear that the expected runtime and number of nodes to search grows exponentially as k increases – regardless of the heuristic’s attempts to slow/linearize the growth.

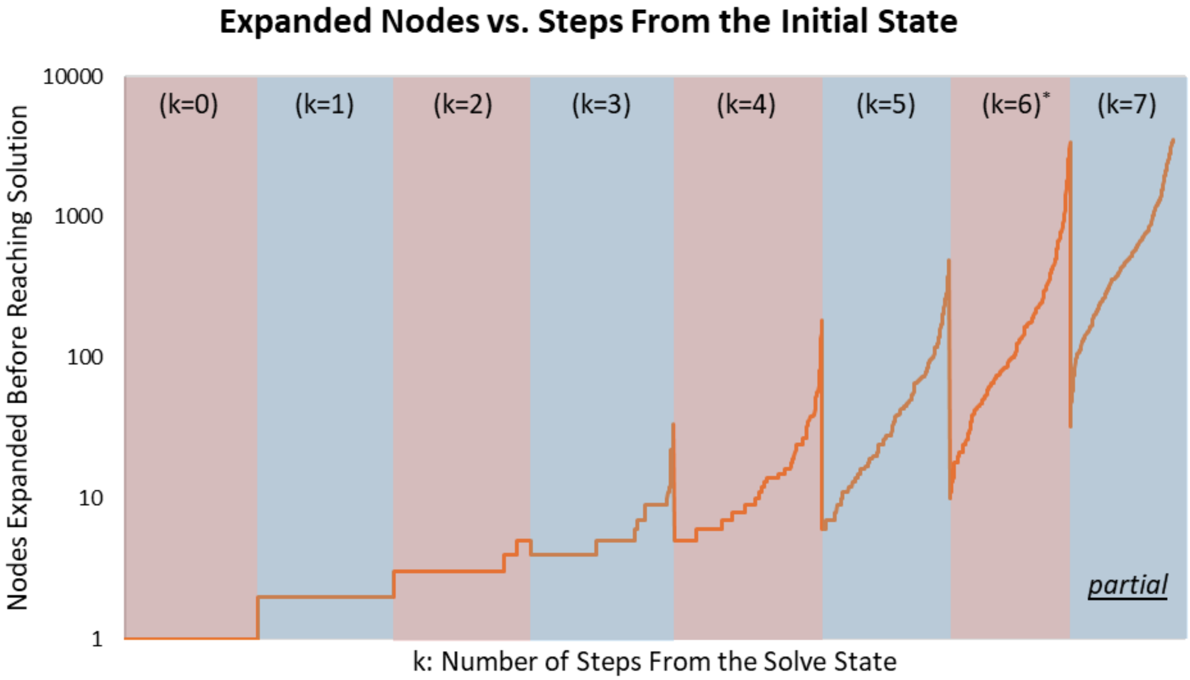


Expected – or average – number of nodes expanded to solve the pyraminx given a solution of k-steps. With a 6th-degree polynomial of best fit (R^2=0.998).



Additionally, we should recognize that – although the expected number of searched nodes remains below 1000 for (k <= 7) – there is significant variation between the smallest and largest node expansions recorded. This is because the heuristic serves as a guesstimate instead of a certain path forward. For instance, some 6-move solutions can take as few as 10 node expansions – if the bit-configuration favors the heuristic – and others can take up over 3000, sometimes 6000 – if the bit-configuration is unfavorable to the heuristic. (See next graph for additional details.)

Note: There was a single run of (k=6) where over 6000 nodes were expanded. It was excluded from the previous graph-set to identify the general relationship between the solution length and number of node expansions. I'd show a trendline, but 1,000 identical outputs of (0,1) and (1,2) seriously hamper its reliability.



** There was a single run of (k=6) where over 6000 nodes were expanded. It was ignored to identify the general pattern present in the expansion.*

As a final piece of data – the node expansions were sorted by their distance from the solve-state and the number of node expansions required to solve the problem, and then the set was mapped logarithmically.

This revealed a divergent, telescoping pattern in the data-set, which is worthy of further analysis by other students in the future.

Additional note: Since potentially 40 runs were removed from the (k=7) experiment, we should note that it is only a partial graph of the required node expansion.

LEARNING OUTCOME:

This is the first time that I have completed a working project in artificial intelligence. Although difficult, and sometimes tedious, I'm proud of the work that I've done, and I am grateful to say that it has given me a practical understanding of A* and more experience in developing user-friendly programs.

Additionally, this project was a massive undertaking – had I planned out my work more clearly before starting, I would have saved myself a great deal of time in preparing my report and completing the implementation. Going forward, I will be sure to build a plan for my work before pushing through blindly.

REFERENCES:

My previous program – and all references used to develop it – remain the largest source for this project. (structure_documentation.txt)

Tony Chiu is responsible for most of the heuristic's pseudocode and design.

https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

^ *Primary resource for writing A* pseudocode.*

https://www.geeksforgeeks.org/what-does-the-if-__name__-__main__-do/

^ *Used to set stand-alone Python programs as modules when necessary.*