CMPT 310
Assignment 1
Jiadi Luo 301354107
Haoxuan Zhao 301385109

Instructions:

- You can do this assignment individually or in a team of two. If you are doing it in a group, only one submission per group is required. Self enrolled groups are available on canvas.
- You may submit multiple times until the deadline. Grade penalties will be imposed for late submissions (see the course outline for the details).
- Always plan before coding.
- For coding questions - Use function-level and inline comments throughout your code. We will not be specifically grading documentation. However, remember that you will not be able to comment on your code unless sufficiently documented. Take the time to document your code as you develop it properly.
- We will carefully analyze the code submitted to look for plagiarism signs, so please do not do it! If you are unsure about what is allowed, please talk to an instructor or a TA.
- Make sure that in your answers, you clearly indicate the exact section you are answering.
- You will submit one .zip file.
    - For theoretical questions - Your submission must be formatted as a single PDF file; other types of submissions (non-pdf, multiple files, etc.) will not be graded. Your name(s) and student number(s) must appear at the top of the first page of your submission.
    - Please put all your files (pdf + code) in one folder. Compress these into a single archive named a1.zip and submit it on Canvas before the due date listed there.

Question 1 [25 marks]: Search Algorithms

Consider the problem of finding the shortest path from *a1* to *a10* in the following directed graph. The edges are labelled with their costs. The nodes are labelled with their heuristic values. Expand neighbours of a node in alphabetical order, and break ties in alphabetical order as well. For example, suppose you are running an algorithm that does not consider costs, and you expand a; you will add the paths <*a,b*> and <*a,e*> to the frontier in such a way that <*a,b*> is expanded before <*a,e*>.

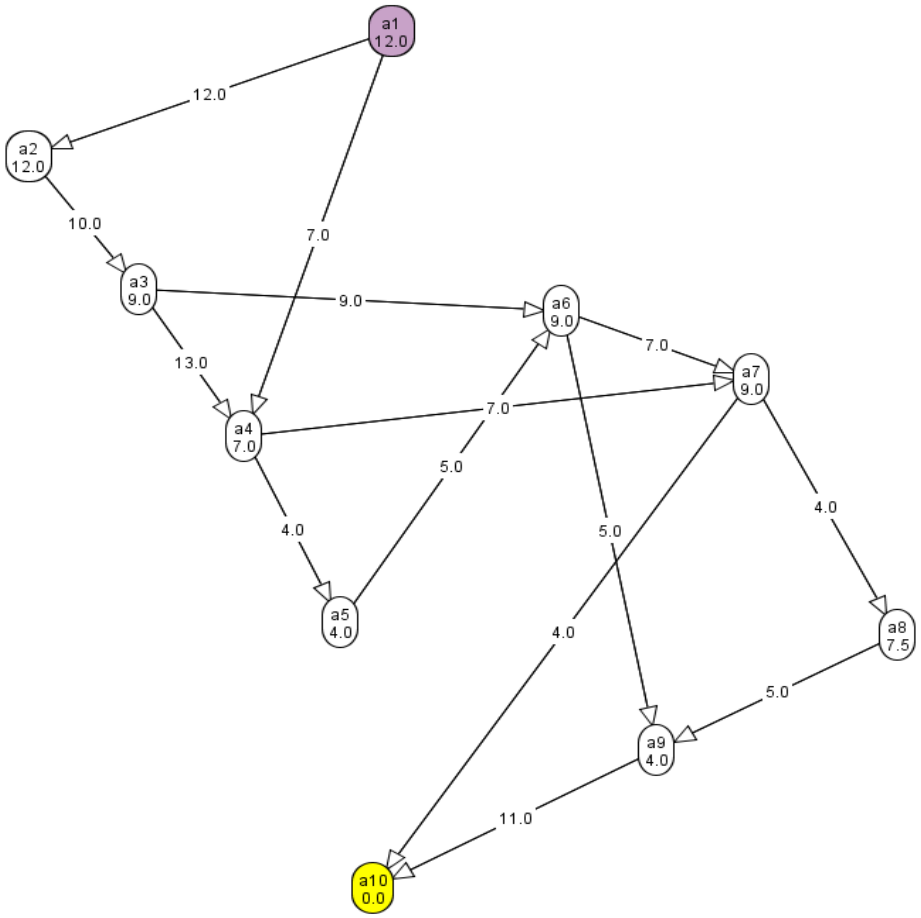


Figure 1: Graph

Note: You may need more rows than given in the table.

(a) [4 marks] Use **breadth-first search** to find the shortest path from *a1* to *a10*, show all paths explored for each step. First two steps are given as an example.

Node explored	path
a1	<a1, a2>, <a1, a4>
a2 (<a1, a2>)	<a1, a2, a3>, <a1, a4>
a4 (<a1, a4>)	<a1, a2, a3>, <a1, a4, a5>, <a1, a4, a7>
a3 (<a1, a2, a3>)	<a1, a2, a3, a4>, <a1, a2, a3, a6>, <a1, a4, a5>, <a1, a4, a7>
a5 (<a1, a4, a5>)	<a1, a2, a3, a4>, <a1, a2, a3, a6>, <a1, a4, a5, a6>, <a1, a4, a7>
a7 (<a1, a4, a7>)	<a1, a2, a3, a4>, <a1, a2, a3, a6>, <a1, a4, a5, a6>, <a1, a4, a7, a8>, <a1, a4, a7, a10>

a4 (<a1, a2, a3, a4>)	<a1, a2, a3, a4, a5>, <a1, a2, a3, a4, a7>, <a1, a2, a3, a6>, <a1, a4, a5, a6>, <a1, a4, a7, a8>, <a1, a4, a7, a10>
a6 (<a1, a2, a3, a6>)	<a1, a2, a3, a4, a5>, <a1, a2, a3, a4, a7>, <a1, a2, a3, a6, a7>, <a1, a2, a3, a6, a9>, <a1, a4, a5, a6>, <a1, a4, a7, a8>, <a1, a4, a7, a10>
a6 (<a1, a4, a5, a6>)	<a1, a2, a3, a4, a5>, <a1, a2, a3, a4, a7>, <a1, a2, a3, a6, a7>, <a1, a2, a3, a6, a9>, <a1, a4, a5, a6, a7>, <a1, a4, a5, a6, a9>, <a1, a4, a7, a8>, <a1, a4, a7, a10>
a8 (<a1, a4, a7, a8>)	<a1, a2, a3, a4, a5>, <a1, a2, a3, a4, a7>, <a1, a2, a3, a6, a7>, <a1, a2, a3, a6, a9>, <a1, a4, a5, a6, a7>, <a1, a4, a5, a6, a9>, <a1, a4, a7, a8, a9>, <a1, a4, a7, a10>
a10 (<a1, a4, a7, a10>)	<a1, a2, a3, a4, a5>, <a1, a2, a3, a4, a7>, <a1, a2, a3, a6, a7>, <a1, a2, a3, a6, a9>, <a1, a4, a5, a6, a7>, <a1, a4, a5, a6, a9>, <a1, a4, a7, a8, a9>, <a1, a4, a7, a10>

Therefore, the shortest path from *a1* to *a10*: <a1, a4, a7, a10>

(b) [6 marks] Use **lowest-cost-first search** to find the shortest path from *a1* to *a10*, show all paths explored, and their costs, for each step. First step is given as an example.

Node explored	Cost	Path
a1	7	<a1, a4>
	12	<a1, a2>
a4	11 14	<a1, a4, a5> <a1, a4, a7>
a5	16	<a1, a4, a5, a6>
a2	22	<a1, a2, a3>
a7	18 18	<a1, a4, a7, a8> <a1, a4, a7, a10>
a6	21 23	<a1, a4, a5, a6, a9> <a1, a4, a5, a6, a7>
a8	23	<a1, a4, a5, a7, a8, a9>
a10		

Therefore, the shortest path from *a1* to *a10*: <a1, a4, a7, a10>

(c) [12 marks] Use **A* search** to find the shortest path from *a1* to *a10*, show all paths explored, and their values of f(n), for each step.

(Note that f(n) = g(n) + h(n), where g(n) is the cost of the path from the start node to the current node n, h(n) is the heuristic value of the current node n)

First step is given as an example.

Iteration	Node(s)	Actual path cost	Heuristic cost	Total cost	Path(s)
0 (<a1>)	a1	0	12	12	<a1>
1 (<a1, a4>)	a4 a2	7 12	7 12	14 24	<a1, a4> <a1, a2>
2 (<a1, a4, a5>)	a5 a7 a2	11 14 12	4 9 12	15 23 24	<a1, a4, a5> <a1, a4, a7> <a1, a2>
3 (<a1, a4, a7>)	a7 a2 a6	14 12 16	9 12 9	23 24 25	<a1, a4, a7> <a1, a2> <a1, a4, a5, a6>
4 (<a1, a4, a7, a10>)	a10 a2 a6 a8	18 12 16 18	0.0 12 9 7.5	18 24 25 25.5	<a1, a4, a7, a10> <a1, a2> <a1, a4, a5, a6> <a1, a4, a7, a8>

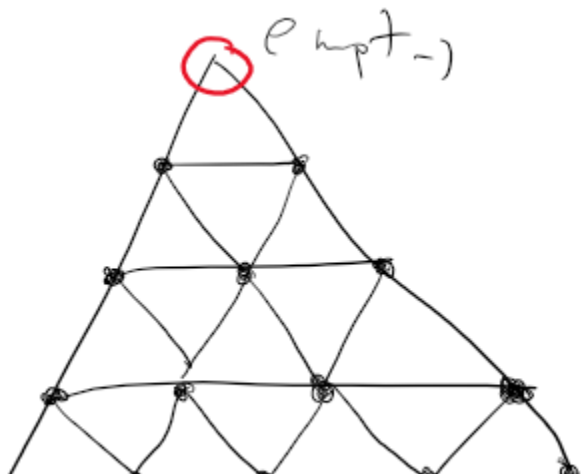
Therefore, the shortest path from *a1* to *a10*: <a1, a4, a7, a10>

(d) [3 marks] Out of BFS, LCFS, A*, which search algorithm do you think is most efficient for the above example? Justify your reasoning.

A* search algorithm is the most efficient one because it applies the fewest iterations and expands the least nodes while the first two algorithms apply more.

Question 2: [25 marks] Game

For this activity, you are going to play the Seashell game. The dots are closed seashells. The rules of the game are to use the seashells to jump over other seashells like in checkers, which in turn will open the seashell that has been jumped over. The seashells can also be moved into the empty spot adjacent to it but will not open any shells. The goal is to do this for all the seashells to win the game.



You can play the game here: https://www.novelgames.com/en/seashell/

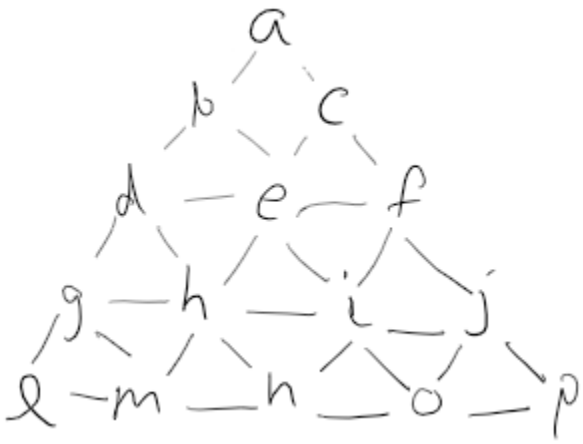


Fig 2. Seashell board

Now, you are going to represent seashells as a search problem. (Use the labels provided in Fig 2 for referring to spaces on the board).

(a) [4 points] How would you represent a node/state?

$x_{00,}$ $x_{10}x_{11}$, $x_{20}x_{21}x_{22}$, $x_{30}x_{31}x_{32}x_{33}$, $x_{40}x_{41}x_{42}x_{43}x_{44}$

$x_{i,j}$ is the open/close state of the seashell in row i and column j. i, j ∈ {0,1,2,3,4}. $x_{i,j}$ ∈ {0,1,-1}. $x_{i,j}$ = 1 denotes the open seashell, $x_{i,j}$ = 0 denotes the closed seashell, and x_{ij} = -1 denotes the empty spot.

For example, the initial state would be:

$x_{00,}$	$x_{10}x_{11}$,	$x_{20}x_{21}x_{22}$,	$x_{30}x_{31}x_{32}x_{33}$,	$x_{40}x_{41}x_{42}x_{43}x_{44}$
-1,	00,	000,	0000,	00000

In other words, we can use a binary string with 15 variables, each represents a node.

(b) [2 points] In your representation, what is the goal node?

The goal node will be the binary string with fourteen 1's and one -1.

(c) [3 points] How would you represent the arcs?

We can represent the arcs by a pair of node indexes, indicating which node the seashell moves to which node. For example, (x_{10}, x_{32}) represents (b, i) in Figure 2 and this means the seashell is being moved from node b to node i by jumping the node e in between them.

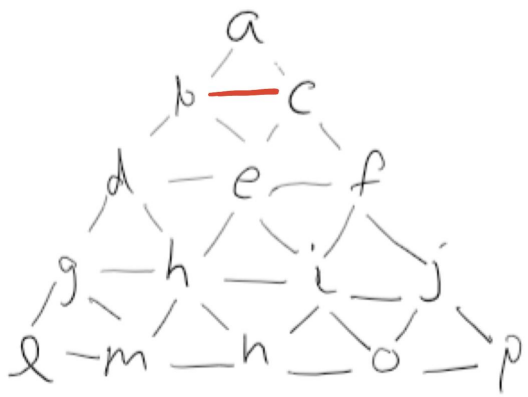
Also, when the seashell $x_{i,j}$ has empty spots around, $x_{i,j}$ can be moved to the node next to it. For example, $x_{2,2}$ = -1 means f is an empty spot, then $x_{2,1}$ e can be moved to f, which can be implemented as (e, f).

Based on this rule, we can:
Move the seashell located at x_{ij} by jumping over its adjacent seashells at node $x_{i,j-1}$ (left), $x_{i,j+1}$(right), $x_{i-1,j-1}$(top-left), $x_{i-1,j}$ (top-right), $x_{i+1,j}$ (bottom-left), and $x_{i+1,j+1}$ (bottom-right) when the empty spot is on the other side. Move the seashell located at x_{ij} to the empty spot beside it.
For example, we can represent $x_{i,j}$ and its possible arcs by using:

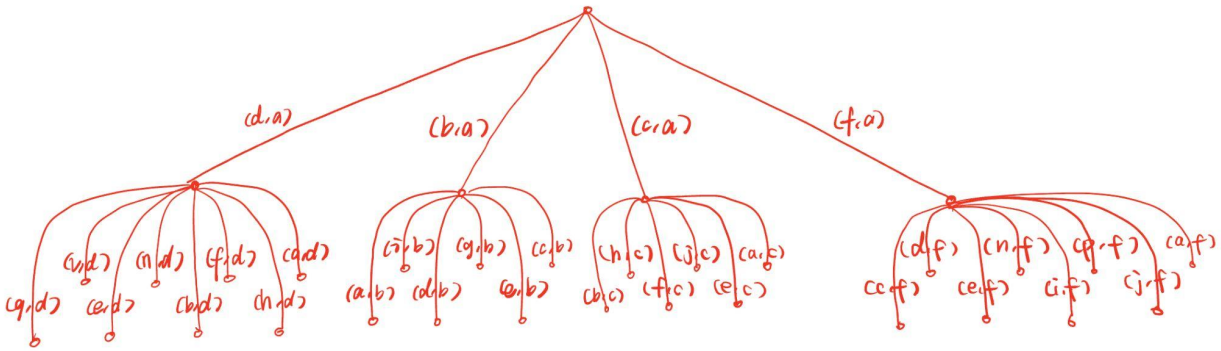
- $(x_{i,j}, x_{i,j-2})$, move seashell at $x_{i,j}$ to node $x_{i,j-2}$ and open/close the seashell $x_{i,j-1}$ in between them (left)
- $(x_{i,j}, x_{i,j+2})$, move seashell at $x_{i,j}$ to node $x_{i,j+2}$ and open/close the seashell $x_{i,j+1}$ in between them (right)
- $(x_{i,j}, x_{i-2,j-2})$, move seashell at $x_{i,j}$ to node $x_{i-2,j-2}$ and open/close the seashell $x_{i-1,j-1}$ in between them (top-left)
- $(x_{i,j}, x_{i-2,j})$, move seashell at $x_{i,j}$ to node $x_{i-2,j}$ and open/close the seashell $x_{i-1,j}$ in between them (top-right)
- $(x_{i,j}, x_{i+2,j})$, move seashell at $x_{i,j}$ to node $x_{i+2,j}$ and open/close the seashell $x_{i+1,j}$ in between them (bottom-left)
- $(x_{i,j}, x_{i+2,j+2})$, move seashell at $x_{i,j}$ to node $x_{i+2,j+2}$ and open/close the seashell $x_{i+1,j+1}$ in between them (bottom-right)
- (x_{ij}, empty spots beside it), move seashell at $x_{i,j}$ to the empty spots beside it.

(d) [3 points] How many possible board states are there? Note: this is not the same as the number of "valid" or "reachable" game states, which is a much more challenging problem.
Each of these 14 seashells can be either closed or opened which is 2^14 possible states; the empty spot can be in any position of this board which is 15 possible states. Therefore, the total possible states would be 15 * 2^14 = 245760.

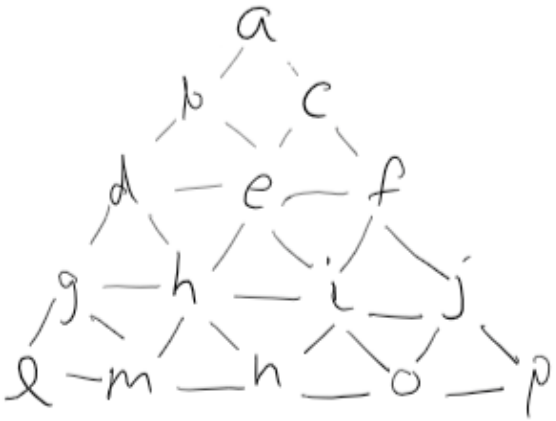
(e) [6 marks] Write out the first three levels (counting the root as level 1) of the search tree based on the labels in Figure 2. (Only label the arcs; labelling the nodes would be too much work).
- So there exists a bug in the Fig2. where b and c should be connected according to the game itself. If we fix this bug the Fig2. will be updated as shown below:



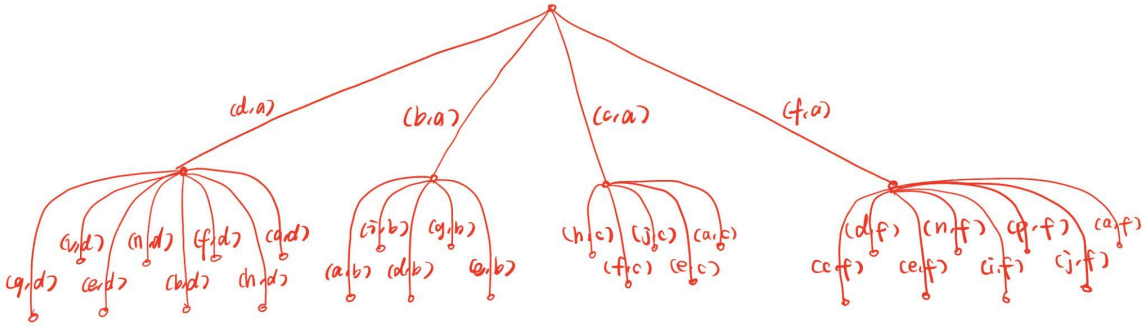
And our answer for this question would be:



- However, if prof wants us to ignore this bug and proceed with the original Fig2.:



Our result would be:



(f) [3 marks]What kind of search algorithm would you use for this problem? Justify your answer.
The problem is uninformed, thus we can choose one search algorithm from IDS, BFS, or DFS. However, DFS cannot be chosen in this case because cycles are possible as there will always be a move that can reverse a state to its parent state. There is no need to use IDS since we know that the optimal solution for this game is 19 steps. Therefore, we can choose BFS.

(g) [4 marks] Would you use cycle-checking? Justify your answer.
Yes, in this example, every state can be reversed to its parent state by one move, and we know that any path containing repeated states cannot be optimal. We can decrease the branching factor for the search tree to improve the performance by using cycle-checking.

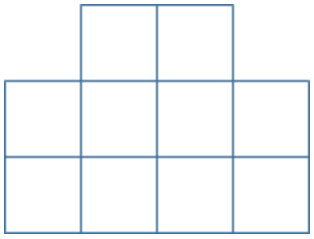
Ques 3. [25 marks] Programming 1

For this question, you get a chance to play with some heuristics. We have provided you with the code needed for this activity. There are two files:

- search.py
- Util.py

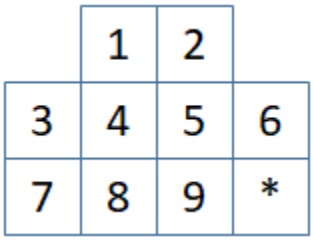
These files are available on canvas. Go through the code and understand it, including the Problem class that it inherits from.

In search.py, we have added a new StagePuzzle class, which is a modified version of the EightPuzzle class. We call our modified puzzle as StagePuzzle since our puzzle is like a champion stage, as shown below.



StagePuzzle

The desired final state of our puzzle is as follows:



StagePuzzle Final State

(a) [5 marks] Write a function called `make_rand_StagePuzzle()` that returns a new instance of a `StagePuzzle` problem with a random initial state that is solvable. Note that `StagePuzzle` has a method called `check_solvability` that you should use to help ensure your initial state is solvable.

(b) [5 marks] Write a function called `display(state)` that takes a `StagePuzzle` state (i.e. a tuple that is a permutation of (0, 1, 2, …, 9)) as input and prints a neat and readable representation of it. 0 is blank and should be printed as a * character.

For example, if `state` is `(0, 3, 2, 1, 8, 7, 4, 6, 5, 9)`, then `display(state)` should print:

```
 * 3
2 1 8 7
4 6 5 9
```

(c) [15 marks] Create 8 (more would be better!) random StagePuzzle instances (using your code from above) and solve each of them using the algorithms below. Each algorithm should be run on the exact same set of problems to make the comparison fair.

For each solved problem, record:

- the total running time in seconds

- the length (i.e. number of tiles moved) of the solution
- that total number of nodes that were *removed* from the frontier

You will probably need to modify the A* function named "astar_search" in the provided code to get all this data.

Note:

- The time it takes to solve random StagePuzzle instances can vary from less than a second to hundreds of seconds. So solving all these problems might take some time!
- The result function in StagePuzzle class does not necessarily check the requested action's feasibly before doing it. It is your responsibility to check the actions feasibilities before doing them. You can also edit the StagePuzzle class and add new functions if you need them.
- Make sure to add the comments in the code.

The algorithms you should test are:

- A*search using the misplaced tile heuristic (this is the default heuristic in the StagePuzzle class)
- A* search using the Manhattan distance heuristic. Please implement your version of the Manhattan heuristic.
    - Be careful: there is an incorrect Manhattan distance function in tests/test_search.py. So, don't use that!
- A*search using the max of the misplaced tile heuristic and the Manhattan distance heuristic

Summarize all your results in a single table for comparing the three heuristic functions you used. Based on your data, which algorithm is the best? Explain how you came to you conclusion.

A: Manhattan distance heuristic B: Misplaced tile heuristic C: Combined heuristic

Initial state	**Total running time**			**Solution length**			**Removed frontier nodes**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
(7, 9, 4, 8, 2, 5, 1, 3, 0, 6)	0.77s	364.04s	3.58s	29	29	29	2368	52248	5503
(8, 0, 6, 3, 5, 9, 7, 2, 1, 4)	3.93s	1289.41s	7.34s	31	31	31	5302	89576	7763
(4, 8, 3, 1, 6, 9, 0, 7, 2, 5)	0.04s	2.74s	0.13s	23	23	23	536	4572	905
(8, 1, 7, 3, 4, 9, 2, 6, 0, 5)	0.01s	0.83s	0.02s	21	21	21	224	2263	302
(4, 8, 0, 9, 5, 1, 6, 3, 2, 7)	0.71s	3063.71s	3.62s	32	32	32	2320	122802	5314
(6, 5, 1, 2, 8, 4, 3, 9, 0, 7)	1.11s	421.46s	3.04s	29	29	29	2739	51257	4765
(1, 4, 0, 3, 5, 2, 8, 9, 7, 6)	1.45s	33.18s	2.08s	26	26	26	3312	16038	3632

(6, 2, 9, 0, 4, 1, 3, 8, 7, 5)	0.08s	22.39s	0.24s	25	25	25	662	12726	1262

Summary of Results: (Calculated using mean values)

Heuristic	Total run time	Solution length	Node expanded
Misplaced tiles	649.72s	27	43935
Manhattan distance	1.01s	27	2183
Max of both	2.48s	27	3681

Your observations:

The Manhattan distance heuristics takes a least amount of total run time and node expanded, thus is the best algorithm here.

The combined heuristic expanded more nodes than Manhattan distance heuristic, which is not possible if both Misplaced tile heuristic and Manhattan distance heuristic implemented for this problem are admissible. Note that the provided Misplaced tile heuristic function also counts the position of the empty spot which is not admissible, i.e. for the state n: (1, 2, 3, 4, 5, 6, 7, 8, 0, 9) that is only 1 step away from the goal state: (1, 2, 3, 4, 5, 6, 7, 8, 9, 0) but the provided misplaced tile heuristic function will have h(n) = 2 which overestimates the actual cost since it also counts if the empty spot "0" is in the goal position.

Below is a sample solution:

Total Results:

A: manhattan heuristic, B: misplaced tiles heuristic, C: combined heuristic

Initial State	Total run time			Solution length			Nodes expanded		
	A	B	C	A	B	C	A	B	C
(4, 9, 3, 6, 0, 2, 7, 8, 5, 1)	1.19s	119s	2.33s	26	26	26	1461	18512	2785
(4, 5, 8, 3, 6, 9, 1, 2, 7, 0)	0.61s	58s	1.03s	26	26	26	941	13318	1802
(1, 7, 5, 4, 8, 2, 3, 9, 0, 6)	0.29s	12s	0.26s	23	23	23	636	6364	854
(8, 9, 7, 1, 3, 6, 4, 5, 0, 2)	3.83s	291s	3.06s	27	27	27	2793	25923	3142
(1, 6, 2, 4, 8, 9, 0, 3, 7, 5)	0.04s	0.38s	0.02s	19	19	19	110	1001	112
(1, 4, 9, 5, 7, 6, 3, 2, 0, 8)	0.65s	59s	1.41s	25	25	25	995	12780	2071
(6, 4, 3, 1, 8, 9, 7, 2, 5, 0)	0.04s	0.19s	0.02s	18	18	18	88	694	120
(6, 4, 3, 8, 9, 0, 2, 7, 1, 5)	0.18s	31s	0.15s	25	25	25	520	10259	670

Summary of Results: (Calculated using mean values)

Heuristic	Total run time	Solution length	Nodes expanded
Misplaced tiles	71.32s	24	11,106
Manhattan distance	0.85s	24	943
Max of both	1.04s	24	1,445

Hints

- When you are testing your code, you might want to use a few hard-coded problems that don't take too long to solve. Otherwise, you may spend a lot of time waiting for tests to finish!

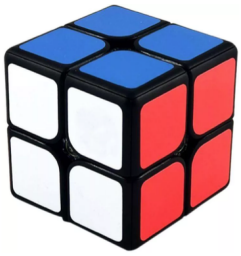
- One easy way to time code is to use Python's standard ``time.time()` function, e.g.::

```
import time

def f():

  start_time = time.time()

  # ... do something ...

  elapsed_time = time.time() - start_time

  print(f'elapsed time (in seconds): {elapsed_time}s')
```

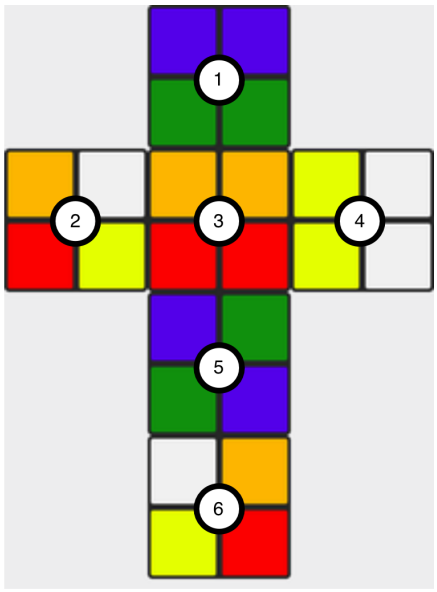
Python has other ways to do timing, e.g. check out the ``timeit`` module.

Question 4: [25 marks] Programming II - You can do (a) in any programming language. TAs would be able to help you with python only.

(a) [20 marks] In this problem, you need to input a 2*2 rubik's cube and solve it using A* searching algorithm. The transition between the states is the same as real moves on a rubik's cube (ie. in each step, one of the facets can be turned clockwise or counterclockwise and there are 12 choices in total). Consider the cube as the following image.



The colors are numbered as Orange=1, Green=2, White=3, Blue=4, Red=5, and Yellow=6. You input the facets of the initial state of the cube with the following order:



For example, the input for the above state is:

4 4 2 2
1 3 5 6
1 1 5 5
6 3 6 3
4 2 2 4
3 1 6 5

You need to output the moves by specifying the number of the facet and its direction (ie. clockwise vs counterclockwise). For example:

Turn Facet#1 Clockwise
Turn Facet#2 Counterclockwise
…

After finding the solution, you should also print the number of explored nodes and depth of the found goal node. Use the following heuristic function for the A* algorithm.

h(n) = 4 * (number of facets with 4 different colors)
+ 2 * (number of facets with 3 different colors)
+ number of facets with 2 different colors

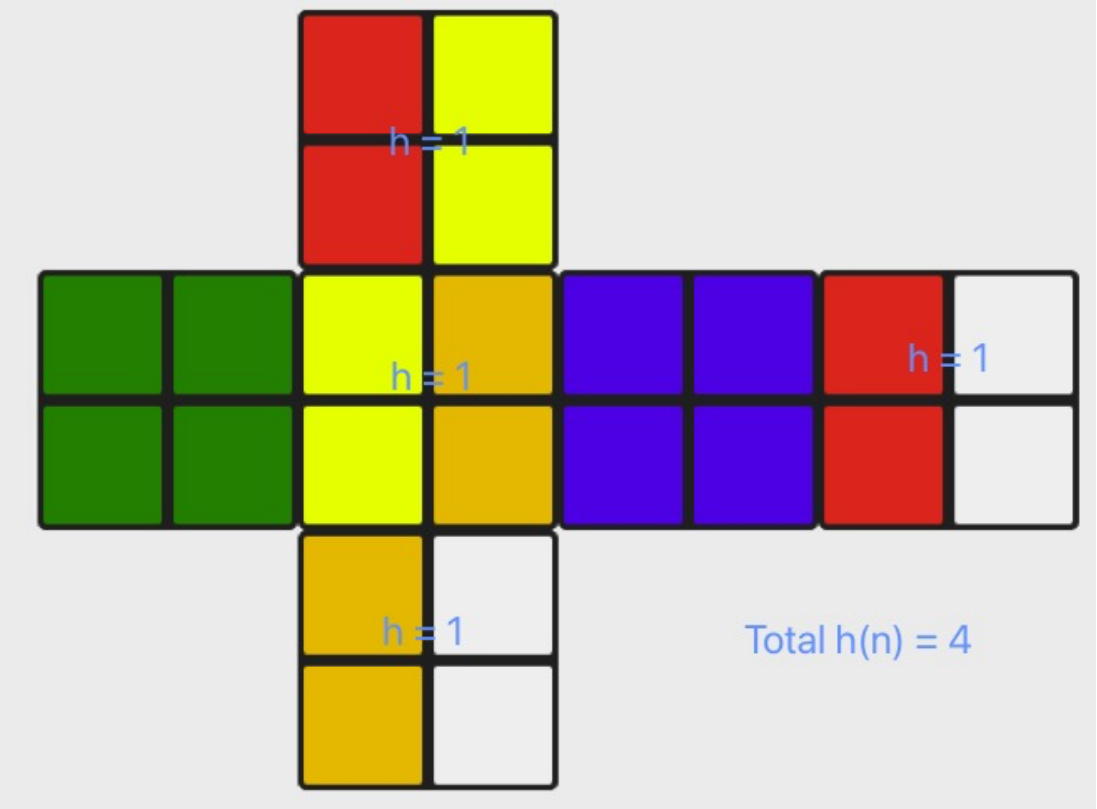
For instance, the value of the heuristic function for the previous example is 12.

Below is a sample output:

```
cube colors:
4 4 2 2
1 3 5 6
1 1 5 5
6 3 6 3
4 2 2 4
3 1 6 5
[[2, 2, 2, 2], [6, 6, 6, 6], [5, 5, 5, 5], [3, 3, 3, 3], [4, 4, 4, 4], [1, 1, 1, 1]]
Produced Nodes: 18191
Expanded Nodes: 4027
Answer Depth: 10
Max Memory: 18191

Solution:
turn facet#1 anticlockwise
turn facet#3 anticlockwise
turn facet#2 anticlockwise
turn facet#1 clockwise
turn facet#2 anticlockwise
turn facet#2 anticlockwise
turn facet#1 anticlockwise
turn facet#2 clockwise
turn facet#3 clockwise
turn facet#1 anticlockwise
```

(b) [5 marks] Do you think that the solutions found by this algorithm are always optimal? why?

No, the heuristic function for this question is not admissible. For example, a counterexample could arise when the cube is 1 step away from the goal state where there are 4 facets with 2 different colors (see image below). In this case, h(n) = 1+1+1+1 = 4 > c(n) = 1, and h(n) overestimates the actual cost which is obviously not admissible. Therefore, A* search algorithm with this heuristic function doesn't guarantee an optimal solution.



Note:

For specific help related to Python - Drop-in TAs programming OH.