Maman 11 – Eran Moshe

# **How to run:**

unzip maman11.zip into working directory  
open terminal in working directory  
pip install requirements.txt  
python tiles.py 8 2 1 5 3 7 4 6 0

requirements.txt:  
numpy==1.26.2

# **Data structures**

PriorityQueue   
implemented using pythons heapq. Implemented put, get and empty (is\_empty). Also added a counter for when we have the same priorities, so first in, first out.

Stack   
implemented using pythons deque (put, get, empty)

Node  
created to represent a searching state and previous action that get us to that state. Have additional parameters like cost (which is not used), solved, …

actioned\_tile – to print the tiles that moved for the solution.

\*\*kwargs – dynamic variables I used when implemented gbfs and a\_star

Parent – the previous node (and state) we came from. Once we have a solution (state = final\_state) these links will help us print the solution (actioned tiles, in reverse)

Do note that the links are ONLY from son (successor) to parent (predecessor). So when finding a solution we get this link and reverse it to print the solution.

*def* get\_path(self) -> list:  
 path = list()  
 curr\_node = self  
 *while* curr\_node:  
 *if* curr\_node.actioned\_tile != *None*:  
 path.append(curr\_node.actioned\_tile)  
 curr\_node = curr\_node.parent  
  
 *return* list(reversed(path))

# **Classes**

Throughout the code we use **numpy** to represent the board/state and do numeric/search operations within the board/state. This is because numpy is highly optimal in dealing with those scenarios. Using python native lists will be highly inefficient and not professional.

# TilesGame

This class is representing a 3x3 tile game. It has two fields:

Winning\_boards – which represented the 2 winning boards which was defined in the maman. But changed to 1 winning board after the forum was updated.

Mults – np array of powers (1, 10, 100, …, 10\*\*8) shape (3, 3) – is used to hash a board / state. Do note that in TilesGame, we say `board`. And in the Agents classes we will say `state`

*def* is\_final\_state(self, *board*):  
 """ return True if board == final state """

*def* get\_winning\_boards(self):

defines the winning boards and save them as part of the TilesGame Object

*def* hash\_board(self, *board*):  
 """ given board returns hashed value """  
 \_hash = np.sum(*board* \* self.mults)  
 *return* \_hash

*def* is\_input\_correct(*input\_str*):

input check

*def* init\_board(*user\_input*):  
 """ init a board game from user string. """

# **“Transition function” – These functions explain the transitions**

*def* get\_valid\_moves(*board*, *prev\_move*='') -> list:

This is part of the “Transition function”. Given a state and the previous move, it gives all legal/valid moves. This includes consideration in board edges and not go against the previous move (if I moved left, no point in moving right).

*def* get\_swapping\_indices(*empty\_cell\_loc*, *move*):  
 """ given empty cell and the move, return the indices (x, y)  
 of the cell needed to be swapped"""

This is the continue of the “Transition function” so that, getting the empty cell location (where 0) and where I want to move it, I get the indices of the tile. (for example, 0 is in (1,1) and I want to move 1 right, I will get (1, 2). Note that empty\_cell\_loc and move MUST BE VALID. All checks of validity happened beforehand.

*def* make\_move(*\_board*, *move*):  
 """ only accept valid moves. validity responsibility is on the user  
 change the board according to the move.  
 returns new board, moved\_tile ""

The final stage of the “Transition Function” Given board and move, we locate the 0 cell. Using the `move` argument we call get\_swapping\_indices. And then we swap the 0 position with the correct tile.

# **HEURISTIC**

The following function is to calculate “distances”, this is actually the “heuristics” given 2 boards (current state, win state)  
Do note that all were being trialed during development. Some of them are not admissible.

The final heuristic is Manhattan distance + Direct reversal (which checks the manh dist and the amount of tiles that are swapped. (for example 0, 1 2 will be 0, 2, 1, so we have 2 and 1 swapped here). After some Analysis, these situations add additional 2 moves (at least) to the heuristic, or specifically, to the Manhattan distance (because you cannot directly move first tile, 1 move and 2nd tile 1 move and they’ll be swapped. In best case, it’ll take additional 2 moves).

This was the chosen heuristic for the assignment as well. There was a lot of fails in order to get there and you can enjoy them by reading the following inline comments.

Even **logic fails**, like the actual calculation of l2 and l1 using the numbers of the tiles :> those mistakes show me what happens when taking inadmissible heuristic. You can see in code the implementations of the distances functions.

Heres the function with inline comments:

*def* calculate\_distance(*board1*, *board2*, *dist\_metric*='l2'):  
 *if* str(*dist\_metric*).lower() == 'l2':  
 """ fail implementation.. this uses tiles values instead of positions """  
 *return* np.sqrt(np.sum(np.power(*board1* - *board2*, 2)))  
  
 *elif* str(*dist\_metric*).lower() == 'l1':  
 """ lol what is this? this is not admissible :> same logic error of values instead of positions """  
 *return* np.sum(np.abs(*board1* - *board2*))  
  
 *elif* str(*dist\_metric*).lower() == 'euc':  
 dist\_mat = euclidean\_distance(*board1*, *board2*)  
 *return* np.sum(dist\_mat)  
  
 *elif* str(*dist\_metric*).lower() == 'diff':  
 *return* np.sum(np.array(*board2* != *board1*).astype('int8'))  
  
 *elif* str(*dist\_metric*).lower() == 'manh':  
 dist\_mat = manhattan\_distance(*board1*, *board2*)  
 *return* np.sum(dist\_mat)  
  
 *elif* str(*dist\_metric*).lower() == 'crazy\_manh':  
 """ not admissible :> """  
 dist\_mat = manhattan\_distance(*board1*, *board2*)  
 dist\_mat = np.where(dist\_mat >= 1, dist\_mat \* 3, dist\_mat)  
 *return* np.sum(dist\_mat)  
  
 *elif* str(*dist\_metric*).lower() == 'impr\_manh':  
 """ This is a deal breaker! """  
 manh\_dist = np.sum(manhattan\_distance(*board1*, *board2*))  
 count = count\_direct\_reversal(*board1*, *board2*)  
 *return* manh\_dist + count \* 2  
  
 *else*:  
 *raise* ValueError(f"{*dist\_metric*} is not supported.")

afterwords, we’ve implemented the agents

*class* TilesAgent():  
 *def \_\_init\_\_*(self, *name*, *state*: np.ndarray, *game*):  
 self.name = *name* self.explored = set()  
 self.counter\_expends = 0  
 self.init\_state = *state*.copy()  
 self.game = *game*

an attempt to define an abstract class. Eventually I’ve decided to use this for shared parameters only.

Name – every agent has its name  
explored – set to contain the hashes of states visited.  
counter\_expends – counts the nodes that were expended per algo  
init\_state – board init position  
game – the game CLASS!

*class* ASTARTilesAgent(TilesAgent):

self.frontier = PriorityQueue()  
self.dist\_metric = *dist\_metric*self.alpha = *alpha*

# **ALPHA**

So I’ve added here a little nice idea which is called alpha. When calculating the priority:

f(n) = h(n) + g(n) alpha come into play in the following:  
f(n) = 2 \* (1 - alpha) \* h(n) + alpha \* g(n))

alpha is set to 0.5 to not “damage” the original a\_star f(n) calculation.

But you can see this alpha weight is actually saying, how much you should consider the heuristic VS how much you should consider the PATH you’ve made so far.

In the extremes if alpha = 0 we have f(n) = h(n) Which is exactly **GBFS**!

If alpha = 1 we have f(n) = g(n) which in this example is exactly **BFS!**

So, me implementing GBFS and BFS was a waste of time (not really, I’ve learnt a lot).

Through all those 3 algorithms I’ve maintained an explored set.

BFS on its own just uses a regular Queue.

GBFS and A\_STAR using Priority Queues.

I think I will skip taking about GBFS and BFS because my A\_STAR with the alpha idea actually made them a private case of it. Although I will add

GBFS – find heuristic that yield optimal solution is IMPOSSIBLE.

Proof: since GBFS is not optimal for any heuristic (both admissible and consistent) it is not guaranteed to find the optimal solution. The only way to find the optimal solution is if I know the EXACT price it’ll cost me to get from state X to final state. If I know that, I know the solution and I don’t need a search algorithm to solve this.   
Last time I looked, the 8 tiles problem require a search algorithm and the solution could not be determined ahead. Even if I’m wrong, and from 1 look of a board there is an algorithm that immediately solve the board (without running the search Agents) then giving me 10 points to find it and use it as heuristic is an understate!

Last but not least, our **IDDFSTilesAgent**

*class* IDDFSTilesAgent(TilesAgent):

So our frontier here is a stack. And we’ve approach for both a recursive solution and iterative one. This is done because in the iterative solution I could not maintained the explored set.

In the solve function:

*def* solve(self, *max\_depth*=30):  
 *for* i *in* range(1, *max\_depth* + 1):  
 #solution = self.dfs\_traverse(i)  
 solution = self.dfs\_recursive\_init(i)  
 *if* solution *is not None*:  
 *return* solution  
 *return None*

One can comment out the approach he don’t want to use.

I’ve set a max\_depth so that if no solution exists, stop after max\_depth iterations.  
Note: Given problem that is deeper than 30, please increase max\_depth, as when I ran locally, I wanted to avoid running forever.

As I’ve said, in the recursive solution that is an explored set maintained which made the algorithm expand fewer nodes. (but not something grandular)

Running examples for IDDFS (recursive/iterative):

Iddfs recursive (with explored set):

8 2 1 5 3 7 0 4 6  
IDDFS  
1144385  
[4, 1, 2, 7, 6, 5, 7, 4, 3, 6, 5, 7, 8, 2, 4, 5, 7, 8, 5, 4, 1]

Iddfs NOT recursive (NO explored set):

8 2 1 5 3 7 0 4 6  
IDDFS  
1681440  
[4, 1, 2, 7, 6, 5, 7, 4, 3, 6, 5, 7, 8, 2, 4, 5, 7, 8, 5, 4, 1]

**Another interesting example:**

tiles.py 8 2 1 5 3 7 4 6 0  
BFS  
116354  
[7, 1, 2, 8, 5, 3, 8, 2, 1, 8, 2, 5, 3, 4, 6, 7, 8, 2, 5, 1, 2, 5, 4, 3]

IDDFS  
4763707  
[6, 4, 5, 3, 2, 8, 3, 5, 4, 6, 7, 2, 8, 1, 2, 8, 5, 4, 6, 7, 8, 5, 4, 3]

GBFS  
1692  
[7, 3, 5, 8, 2, 1, 3, 5, 8, 4, 6, 8, 4, 2, 1, 4, 5, 3, 4, 1, 2, 5, 3, 4, 1, 3, 5, 2, 3, 1, 4, 5, 2, 3, 1, 2, 5, 7, 8, 5, 7, 4, 2, 1, 3, 7, 5, 6, 7, 3, 1, 2, 4, 5, 2, 1, 3, 2, 5, 4, 1, 3, 2, 5, 3, 1, 4, 3, 5, 2, 1, 4, 3, 5, 4, 3, 5, 4, 2, 1, 3, 2, 4, 5, 2, 3, 1, 4, 3, 1, 4, 3, 6, 7, 3, 4, 1, 6, 4, 3, 7, 4, 6, 1, 3, 6, 4, 7, 6, 3]

A\_STAR  
2245  
[7, 1, 2, 8, 5, 3, 8, 2, 1, 8, 2, 5, 3, 4, 6, 7, 8, 2, 5, 1, 2, 5, 4, 3]

**BFS and IDDFS both find optimal solution but** **ITS NOT THE SAME SOLUTION :>**

**GBFS very fast, expand fewest nodes but solution is insane.**

# Final notes:

Working on this took a lot of hours and was challenging. It took a lot of time, but I enjoyed it. Code is heavily documented and describe everything in detail. Any questions, clarifications etc. I can be contacted here: ernmse@gmail.com