15 Puzzle Problem

Ryan Hiltabrand

Getting Started

This is an artificial intelligence program which attempts to solve the 15 puzzle problem. The program is written in python3. The program can solve the 15 puzzle problen in three different ways.

- Breadth First Search: Expansion of tree until solution is found. Uses a queue to achieve the solution.
- Depth-first search: Expansion down a trees branch until a soulution is found. If not found on branch it moves to the next branch over. Use a stack to achieve this solution
- Informed search using heuristics: Takes the manhattan distance and amount of discplaced tiles and solves the ones with the lowest scores first. Uses a dictonary which finds the lowest score in O(1) time.

This program uses a lot of memory as it runs untill it finds the solution. The 15 Puzzle problem has 1307674368000 combinations meaning that when using an uniformed search like bfs and dfs the program will lead to high ram use as every state expanded is stored so it is not searched over again. As prof of work I implememted the 8 puzzle problem to show that given enough time and ram my algorithm solves the 15 puzzle problem.

Prerequisites

The program was written and tested with python 3.9 but should be able to run on any verision > 3.7. There are three different external libraries used: numpy, timeit, and tracemalloc. Install with the following commands:

- Python Version > 3.7
- python Packages
    - numpy (Arrays)
    - timeit (Track Time)
    - tracemalloc (Track Ram)

Instructions

1. Download and unzip folder
2. Navigate to the directory with main.py
3. Install required packages `pip3 install [package]`
4. Execute code through your operating systems terminal
    - Windows `python3 ./main.py`
    - Linux/Mac `python3 main.py`
5. Follow instructions on screen.
6. Find output in results directory

Program Execution

This program can be run in either 8 puzzle mode or 15 puzzle mode and relies on user input to run.

1. Asks user for which puzzle it wants the AI to solve
    - 8
    - 15
2. Asks the user for which search it wants to conduct
    - B
    - D
    - H
3. Solves the puzzle
4. Stores explored boards and results inside if results directory
5. Explores and writes to file as code executes untill finished

Analysis

Informed search provides as well as the least amount of ram usage. To achieve reviewable results on the 15 puzzle problem I set the board instead of the randomly generated board when I run the program normally [[1,2,3,4], [5,6,7,8], [9,10,11,12], [0,13,14,15]]. Using this basline it appears that heuristics and depth first search are nearly identical as they compute in under .004s. This is as the starting puzzle board is set as an easy to solve by all 3 algorithms. We can however see that bfs will always take the most time as it creates all nodes of the puzzle board before doing the next move. BFS on this test case runs in about .008 s. When looking at memory usage again Heuristics and DFS are similar in this small test case but BFS is nearly double. While I was never able to run a test of a puzzle which was extremly different than the goal as I did not have enough ram to run my code to succession I did try it on another test case on 8 puzzle using my same algorithm and you can see that when using the 3 different types that heuristic solves in .37 s while dfs solves in 2606 s and bfs takes much longer then this. The nodes explored also give great insight into the time it takes to solve the puzzles as dfs explored 241921 while heuristic explored only 562. BFS again explores so many more as it builds a full tree one line at a time.

Considering this fact I believe that while I did not solve the puzzle in the assignment as I have RAM constraints and I can not limit memory as we have to store every combination so we do not go to a previous board state and there are over 100 billion combinations using an AI that does dfs or bfs is not a good idea as searching for the solution just takes so much time to build.

Notes

> - If you want to set a puzzle go into main.py and into the function runner() on line 21.
>
> 1. Comment out the line (line 33) that creates variable board

2. Create a new board variable like the following: `board = np.array([1,2,3,4,5,6,7,8,9,10,11,12,0,13,14,15]).reshape(4,4)`