CS 7320

Homework 2: Search (rev. 9.13.22)

Part 1. Graph Implementation (Dictionary) of node list (5 pts)

1. Write a function to create a python dictionary implementation of a directed/undirected graph from a node-list
2. Use the code developed in (1.) to measure the performance of graph search with BFS and DFS (part 2)

The hw2.zip files contains:
(a) genGraphAsDict.py
(b) test_genGraphAsDict.py
(c) BfsDfs.py

`In the zip file for hw2.zip move the following files to the same directory:`

- genGraphAsDict.py
- test_genGraphAsDict.py

Coding:

- `In genGraphAsDict.py, write code for` **`build_graph_as_dict(..)`** `that builds a Python dictionary implementation of both directed and undirected graphs.`
- Run the test program: test_genGraphAsDict.py to be sure your implementation is correct
- ***Add your code: genGraphAsDict.py to PDF1 under a header Part1***
- ***Add the output of the test program to PDF1***

Then**:**

- `Use this code for part 2`

Part 2. Compare BFS vs. DFS (5 pts)

Compare the performance of the provided BFS and DFS code for finding the shortest path in a graph

Much of the code is written. All you need do here is use your function from part 1 and add a few lines of code to count the number of nodes (boards) visited and run the code

- Download the code for **BfsDfs.py** from the zip file for Homework 2
- Add the function **build_graph_as_dict** from part 1 to your code
  - When you generate your graph dict, use ***`directed=False`*** as in:
  -
  - `graph = build_graph_as_dict(node_list, directed=False)`
- Modify the code to display the shortest path and number of nodes visited for both BFS and DFS
- Add the output of the program to PDF1 under a header Part 2

- Prepare a table showing the number of nodes explored for BFS vs. DFS.
- Add the table to PDF1 under the header Part 2 – no need to include the code.
- `note: be sure to understand what's happening behind the scenes with the line of code:`

```
          list(dfs_all_paths(graph, start_node, goal_node))
```

Part 3. Refactor the BFS algorithm and create a new file: bfs_shortest_path.py (3 pts)

(a) When you run the BFS and DFS algorithms, all the paths to the goal are returned. However, since the first solution found will be the shortest path, there is no need to continue the search. In this section you will use the code from BfsDfs.py to create a new program that only does BFS and returns only the shortest path.

- Set up a new python file: **bfs_shortest_path.py**
- Add your function: build_graph_as_dict
- Add your code for the function: bfs_all_paths – rename the function bfs_shortest_path
- Modify **`bfs_shortest_path (graph, start, goal) so that`** that does NOT return all the paths, only the first found shortest path. This will only require changing of one line of code (hint: look at yield!)
- Output the shortest_path to validate your work. No need to capture output.

(b) Note that the BFS algorithm has a variable `queue` which is list but that acts like a queue since at each iteration the first element is removed using queue.pop(0) as in:

```
        (vertex, path) = queue.pop(0)
```

- In the BFS code provided, a tuple with two values (vertex and path) is stored in the `queue`. However, since the vertex we want to examine as a possible goal will always be found at the end of the path, we only need to store the path in the queue.
- Your task in part b is to modify the code for **bfs_shortest_path** so that **only the path is stored in the queue**. Then, obtain the vertex from the end of the path (list). Your code should look like:

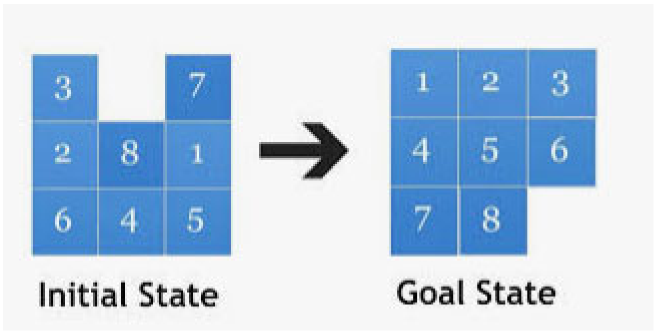
```
        path = queue.pop(0)
        vertex = ..extract next node to examine from path
```

- Test your code to prove to yourself that it works.
- Use you**r bfs_shortest_path** in part 4.
- The reason we are refactoring is to make the code in part 4 easier to write.

Now you are ready to apply this to a real problem

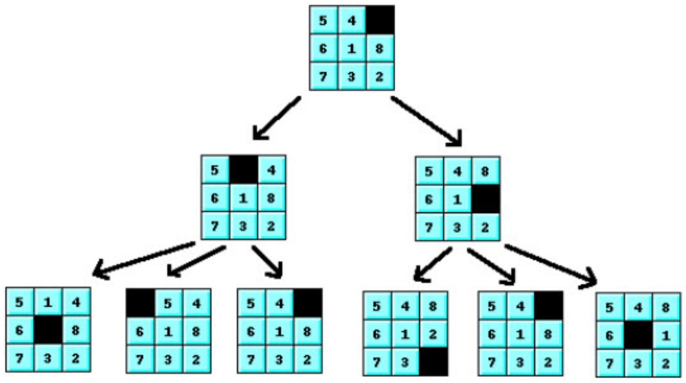
Part 4. The 8 Puzzle : BFS vs. BFS with Heuristic (12 pts)

In the 8-puzzle, there are 8 tiles, numbered 1 through 8, which slide around on the board. In the initial state, the tiles are scrambled. The goal is to put the numbers in order, with the blank square in the lower right hand corner. (a) An initial state. (b) The goal state.



A puzzle board may be represented as a 2D Python array (a list of lists). The blank is represented by a zero. The following is the goal state:
[[1,2,3], [4,5,6], [7,8,0]] where 0 is the blank tile.

As in many real world problems, we are not provided a node list from which we can get child nodes. Given any board we need to generate the child boards (all the next possible boards). The following shows a partial search tree. Note that boards will have 2,3 or 4 children depending on the position of the blank (0).



Write the code to solve the puzzle using BFS.

The key to solving the 8 Puzzle is identify all the child boards of some board, i.e. what moves can be made given some board.

A. Implement a function **`get_child_boards_list(board)`** that returns a list of all the child boards for any given board. Your code should work for any board size, not just 3x3.

For any node, the child nodes will be a list of possible next boards.

For example:

```
[ [1,2,3], [4,0,6], [7,8,5] ]  -> children:[   [ [1,0,3], [4,2,6], [7,8,5] ] ,
                                               [ [1,2,3], [4,8,6], [7,0,5] ],
                                               [ [1,2,3], [4,6,0], [7,8,5] ],
                                               [ [1,2,3], [0,4,6], [7,8,5] ]
                                           ]
```

B. Write a program called **bfs_puzzle.py**, using the above function and applying your refactored BFS algorithm from Part 3 to solve the 8 Puzzle.

- Create a function called **bfs2 (start_board, goal_board)** that executes your BFS algorithm
- Use this function in your BFS code. Your queue will be a list of paths from the start board to some board, eventually the goal_board.
- When you find a board that matches the goal board, the path to it will be the solution to the puzzle.
- Keep track of the number of boards explored.
- DANGER: a possible downside with the 8 puzzle is going into a loop. To avoid looping, only add a board to a path if is not already in the path list.
- Your objective is to find a solution to the start position:

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

Write the code to solve the puzzle using BFS and the Manhattan heuristic (a kind of A*)

- Modify your BFS code from part 4 and add a function **astar(start_board, goal_board)** where you use a heuristic and a priority queue to find a solution to the 8 Puzzle. This amounts to the A* algorithm but without the formality of the f, g and h functions.
- In your BFS with heuristic, replace the list that keeps track of future boards, to a PriorityQueue (either one you write, or the Python PriorityQueue class. Set the priority for each board by computing the Manhattan distance from the board to the goal board. This may be done by totaling the Manhattan distance for each tile to its final destination.
- The heuristic is the **Manhattan distance** (see below) between any given board and the goal board. The value of the Manhattan distance will be value to use for the priority Queue. With a priority queue you will always get the board with the minimum total Manhattan distance to the goal.
- Keep track of the number of boards examined before a solution is found.

- *Report the boards examined for BFS and BFS with Manhattan distance heuristic. What is the percentage improvement in number of boards visited.*

Submit in PDF2
- Table showing comparison
- Code for your bfs_puzzle.py

EXTRA CREDIT

Part 6. Compare the performance of the BFS with heuristics using 4 different distance measures, for the solution of the 15 puzzle. (5 pts)

The distance measures are:

- Manhattan
- Euclidean
- Minkowski
- Hamming

Compare and contrast performance. For Minkowski come up your best p parame**ter**

Solve the problem for the board:

```
b1 = [[2, 3, 7, 4], [1, 6, 8, 12], [5, 9, 11, 15], [13, 10, 0,
14]]
```

where the goal is the board:

```
goal = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14,
15, 0]]
```

Submit:

1. PDF1.
   - The code for genGraphAsDict plus output
   - The table from Part 2.
   - The code for bfs_shortest_path.py followed by the output when testing with the graph in part3.
2. PDF2
   a. The code for bfs_puzzle.py followed by the output from the bfs and the heuristic (A*) algorithms.
   b. Compare the performance for both approaches.
3. A zip file with source code for bfs_shortest_path.py and bfs_puzzle.py

For extra credit, submit PDF3 with code and comparison of results bfs vs. bfs for the 15 puzzle with heuristic, and include code in zip file.

========== Grading based on above points specified
Be sure that the following style specs are followed:

Name&ID in code
Code Format:– no line wrap; no pasted screen shots; courier font; one line of space before for and while loops; no dark backgrounds when presenting code; line numbers appreciated

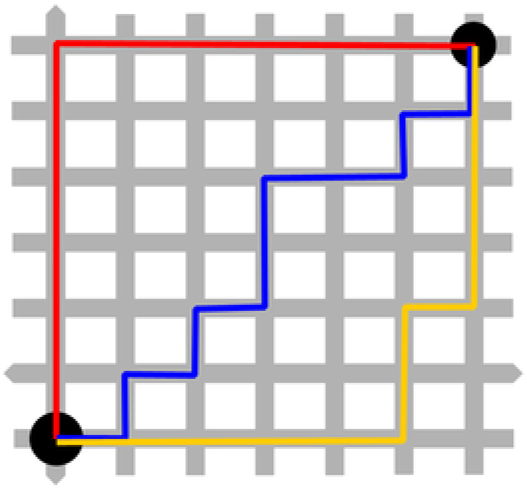
Variable Naming – use Python naming convention, words separated by underscore (my_queue); all names should reflect function
Comments – explain any code that may not be obvious to someone refactoring your code
Well-organized and structured solution
Graphs and tables well presented

Total (25)

Manhattan Distance

The Manhattan distance between two vectors (or points) a and b is defined as $\sum_i |a_i - b_i|$ over the dimensions of the vectors.

This is known as Manhattan distance because all paths from the bottom left to top right of this idealized city have the same distance:



In 2 dimensional plane: It simply is the sum of horizontal and vertical distance between two points,

Consider it between point A(x1,y1) and B(x2,y2):

$$Manhattandistance = abs(x1 - x2) + abs(y1 - y2)$$

The same can be obtained for any dimensions.