

Rush Hour

Searching for Solutions

In the previous part, we implemented key components of a solver for the puzzle game *Rush Hour*. In this part, we continue with this implementation and significantly extend it to find puzzle solutions using various methods.

Please start with your code from part 1 and extend it as directed below. As before, the code for this part should be written in Python to run on tux we will use the same **run.sh** shell script for testing. Again, **you may only use built-in standard libraries (e.g., math, random, etc.); you may NOT use any external libraries or packages**

Paths

Implement a new class, **Path**, that represents a sequence of board states. Add whatever methods are useful for implementing the functionality below, which will likely include **add()** (to add a board to the path), **clone()** (to clone the path for branching), **last()** (to return the last board), and whatever else you need for the rest of the assignment.

Also, please include a path-printing function that prints boards as described in the last part, with one exception: it should print a maximum of 6 boards in a line, and if there are more than 6 boards, it should print the first 6 boards and then continue the rest on a new row of boards (examples below).

Random Walk

Write a method for your **Board** class that does a random walk through next board states. Specifically, given a positive integer $N=10$, the random walk should do the following:

- generate all the moves that can be generated in the board,
- select one at random,
- execute that move,
- and stop if we've reached the goal or we've already executed N moves, otherwise repeat.

The function should return the path generated by the random walk, starting with the first given board.

Add a "**random**" command-line command to your code and print the resulting sequence of boards using the print function written for part 1. It should work on a given board or, if there is no board argument, on the default board (see part 1). Here are some examples:

```
> sh run.sh random
```

```
-----
|  o aa| |  o aa| |  o aa| |  o aa| |  oaa | |  oaaq|
|  o   | |  o   | |  o  q| |  o  q| |  o  q| |  o  q|
|xxo   | |xxo   | |xxo  q| |xxo  q| |xxo  q| |xxo  q|
|ppp  q| |ppp  q| |ppp  q| |pppq| |pppq| |ppp  |
|      q| |      q| |      | |      | |      | |      |
|      q| |      q| |      | |      | |      | |      |
-----
```

```

-----
| oaaq| | oaaq| | oaa | | o aa|
| o q| | o q| | o q| | o q|
|xxo q| |xxo q| |xxo q| |xxo q|
| ppp | | ppp | | pppq| | pppq|
|      | |      | |      | |      |
|      | |      | |      | |      |
-----

```

```
> sh run.sh random " oaa | o | o xx| pppq| q| q"
```

```

-----
| oaa |
| o   |
| o xx
| pppq|
|      q|
|      q|
-----

```

```
> sh run.sh random " oaa | o | oxx | pppq| q| q"
```

```

-----
| oaa | | o aa| | o aa|
| o   | | o   | | o   |
| oxx  | | oxx  | | o xx
| pppq| | pppq| | pppq|
|      q| |      q| |      q|
|      q| |      q| |      q|
-----

```

Note that, because this is a random walk, the code will do different things for each run. In the last example, the random walk was lucky in finding the right move to the solution; often, it's not so lucky, and goes all 10 moves without finding the solution.

Breadth-First Search

Write a method for your **Board** class that does a breadth-first search from the given board, returning the first path found that reaches the solution state. Add a "**bfs**" command-line command that allows a user to perform the BFS on a given board, or on the default board (if there is no board argument). The output should print the path being examined at each step. Here is an example:

```
> sh run.sh bfs
```

```

-----
| o aa|
| o   |
|xxo
|ppp q|
|      q|
|      q|
-----
-----
| o aa| | oaa |
| o   | | o   |
|xxo   | |xxo
|ppp q| |ppp q|
|      q| |      q|
|      q| |      q|
-----

```

```

-----
|  o aa | |  o aa |
|  o   | |  o   |
|xxo    | |xxo    |
|ppp  q| |ppp  q|
|      q| |      q|
|      q| |      q|
-----
|  o aa | |  o aa |
|  o   | |  o   |
|xxo    | |xxo    |
|ppp  q| |ppp  q|
|      q| |      q|
|      q| |      q|
-----
|  o aa | |  o aa |
|  o   | |  o   |
|xxo    | |xxo  q|
|ppp  q| |ppp  q|
|      q| |      q|
|      q| |      |
-----

```

*... and the rest of the paths searched,
ending with the solution and number of paths explored ...*

```

-----
|  o aa | |  oaa | |  oaaq| |  oaaq| |  aaq| |  aaq|
|  o   | |  o   | |  o  q| |  o  q| |    q| |    q|
|xxo    | |xxo    | |xxo  q| |xxo  q| |xx  q| |xxq
|ppp  q| |ppp  q| |ppp   | |ppp   | |opp| |opp|
|      q| |      q| |      | |      | |  o  | |  o  |
|      q| |      q| |      | |      | |  o  | |  o  |
-----
|  oaaq| |  oaaq| |  oaa | |  oaa |
|  o  q| |  o  q| |  o   | |  o   |
|  oxxq| |  oxxq| |  oxx  | |  o  xx
|  ppp | |  ppp | |  pppq| |  pppq|
|      | |      | |      q| |      q|
|      | |      | |      q| |      q|
-----

```

98

A* Search

Write a method for your **Board** class that does an A* search from the given board, returning the first path found that reaches the solution state. Note that as part of this process, you will need to choose and implement an *admissible* heuristic function $h(n)$, such that A* can reasonably estimate the minimum cost from a given state to the goal state.

Add an “**astar**” command-line command that allows a user to perform the search on a given board, or on the default board (if there is no board argument). Here is an example:

```

> sh run.sh astar
-----
|  o aa |
|  o   |

```

```

| x x o
| p p p   q |
|         q |
|         q |
|-----|
|   o  a a | |   o a a |
|   o      | |   o      |
| x x o      | x x o
| p p p   q | | p p p   q |
|         q | |         q |
|         q | |         q |
|-----|

```

*... and the rest of the paths searched,
ending with the solution and number of paths explored ...*

```

|-----| |-----| |-----| |-----| |-----| |-----|
|   o  a a | |   o a a | |   o a a q | |   o a a q | |   a a q | |   a a q |
|   o      | |   o      | |   o   q | |   o   q | |         q | |         q |
| x x o      | x x o      | x x o   q | x x o   q | x x   q | x x q
| p p p   q | | p p p   q | | p p p   | |   p p p | | o p p p | | o p p p |
|         q | |         q | |         | |         | |   o   | |   o   |
|         q | |         q | |         | |         | |   o   | |   o   |
|-----| |-----| |-----| |-----| |-----| |-----|
|-----| |-----| |-----| |-----|
|   o a a q | |   o a a q | |   o a a | |   o a a |
|   o   q | |   o   q | |   o      | |   o      |
| o x x q | | o x x q | | o x x   | | o   x x |
|   p p p | |   p p p | |   p p p q | |   p p p q |
|         | |         | |         q | |         q |
|         | |         | |         q | |         q |
|-----| |-----| |-----| |-----|

```

64

Depending on your chosen heuristic, your number of paths searched may differ from the number in this example, but should be less than the number for BFS.