CMSC 170 Exercise 2

Comparing BFS and DFS

Kobe Jee B. De Luna
2015-06683

2nd Semester A.Y. 2017-2018

1 Comparison of the performance of Breadth-First Search (BFS) and Depth-First Search (DFS).

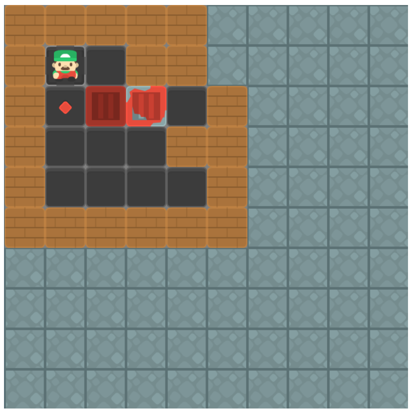
Breadth first search (BFS) and Depth first search (DFS) are both greedy algorithms in a sense that both of the two utilizes a brute force search. In this section, we will be comparing and analyzing the differences of the process that these two algorithms use.

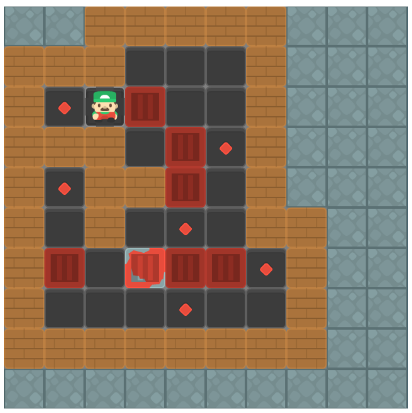
The main difference between the two algorithms is the data structure that they are using. BFS uses a queue, an abstract data type that has FIFO (First in First Out) policy, while DFS uses a stack, an abstract data type that has LIFO (Last In First Out) policy. In terms of the output produced by the two algorithms, BFS always tends to find a solution, if it exists, while in DFS, if the subtree of the node we are traversing is of infinite length, it is possible that it may not find the solution. Also in terms of performance, DFS has proven itself to be more memory efficient than the BFS. To further prove this point, we can safely assume that the binary tree that we have inserted to our data structures will produce a complete binary tree. If we have a complete binary tree that has a depth of 10, all in all, the number of nodes that our tree has is around 2047. Thus, if we searched to our binary tree breadth-wise, we will be needing to traverse more nodes than searching the binary tree depth-wise. Thus, we can say that a complete binary tree is wider than it is tall, making DFS more memory efficient than BFS.

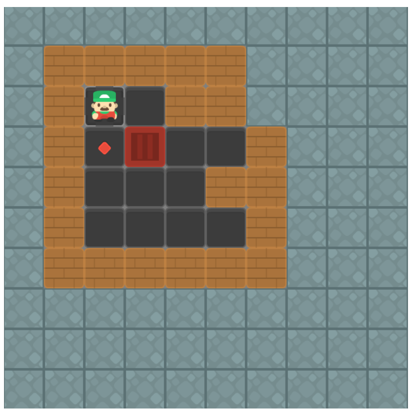
Top left corner's deadlock implementation using Java

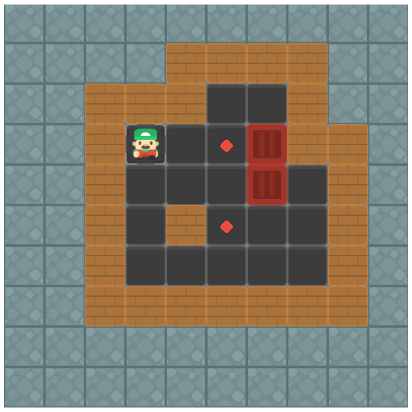
```
 if(clonedArray2D[nextBoxRow-1][nextBoxColumn].equals(World.WALL)
&& clonedArray2D[nextBoxRow][nextBoxColumn-1].equals(World.WALL)){
                        return null;
  }
```

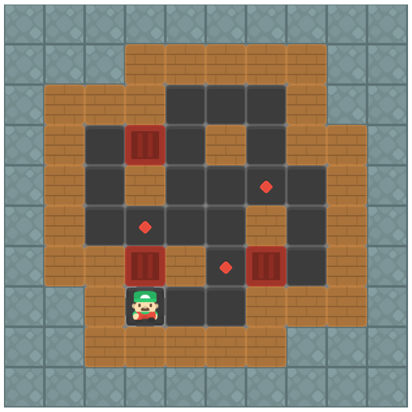
One of the deadlock conditions that I used in implementing both of the two searching algorithms is checking whenever the result of a certain action will lead for one of the box to reach a corner. Since the number of boxes is equal to the number of storage, it can be implied that if a box reached a corner (top-left, top-right, bottom-left, and bottom right), without it being stored in a storage, is already a deadlock state. This certain implementation, along with the the use of explored list, greatly improved the performance of these searching algorithms.


(a) Figure 1


(b) Figure 2


(c) Figure 3


(d) Figure 4


(e) Figure 5

Testcase	**BFS time**	**DFS time**	**BFS moves**	**DFS moves**	**BFS nodes**	**DFS nodes**
Figure 1	20 ms	7 ms	12	12	149	66
Figure 2	1670 ms	4918 ms	34	86	122761	847430
Figure 3	8 ms	1 ms	6	8	117	36
Figure 4	45 ms	117 ms	31	111	3493	2290
Figure 5	29 ms	8 ms	69	81	2274	631

Table 1: Comparison between BFS and DFS

2 What is the more appropriate approach to solving the Sokoban Game: BFS or DFS? Justify your answer.

Since the state space of a Sokoban game is finite, the environment is benign, and the game is fully observable, the more appropriate searching algorithm is depth-first search. DFS, as proven by the results of the testcases (See Table 1), is really memory-efficient, as it generated most of the time fewer number of nodes. It will also be unlikely for the game to have an infinite number of nodes inside a subtree. In terms of the running time, I think both of these two searching algorithms are equal, meaning that no searching algorithm has the precedence over the other in running time.