Using DFS, BFS, UCS search algorithms for solving Sokoban

Le Thi Lien
Student ID : 21522282

(March 30, 2023)

1. How has Sokoban game been modelized?

The game of **Sokoban** can be considered as a search problem. Each map represents a *level*, where boxes are randomly placed. A character - *Sokoban* has to push the boxes around the map so that all boxes are on goals at the end of the game.

The inputs for modeling consist of the following characters: “#”, “ ”, “&”, “X”, “.” and “B”. Each of these characters have a special attribute of the game assigned to it: “#” is a wall, “ ” is a free space, “B” is a box, “.” is a goal place, “X” is a boxes placed on a goal and “&” is the initialized position for Sokoban .

The *initial state* consists of the layout of the game board and the positions of the boxes and the player.

The *goal state* is reached when all boxes are placed in their designated storage locations.

The *state space* is defined by the positions of the boxes and the player on the game board.

Legal actions include moving the player in the four cardinal directions up, down, left, right and pushing a box up, down, left, right. These actions are limited by the position of the player and boxes, as well as constraints such as not moving through walls or over other boxes..

The *successor function* takes a state and an action as inputs and returns the resulting state after performing the action in the given state.

2. Applying DFS, BFS and UCS to Sokoban

2.1. ***Depth First Search***

DFS (Depth-First Search) is a search algorithm that starts from the root node and explores each branch of the search tree as deeply as possible until either a solution is found or there are no more paths to explore.

It maintains a frontier, which is a deque that stores lists of states. Each list represents a path from the initial state to the current state. DFS also maintains a set of explored states to avoid exploring the same state twice. It uses the **legalActions** function to generate a list of legal actions for the current state, and for each legal action, it generates a new state by calling the **updateState** function. If the new state is not a failed state, it adds the new state to the frontier and appends the action to the actions list. The algorithm stops when it finds a state that satisfies the **isEndState** function, and it returns a list of actions that lead to that state.

2.2. ***Breadth First Search***

BFS (Breadth-First Search) is a search algorithm that explores all the neighbors of a node before moving on to the next level of nodes in the search tree.

It is similar to DFS, but it uses a queue instead of a stack to maintain the frontier. It also maintains a queue of actions that correspond to the states in the frontier. The algorithm stops when it finds a state that satisfies the **isEndState** function, and it returns a list of actions that lead to that state.

2.3. ***Uniform Cost Search***

UCS (Uniform-Cost Search) uses similar concepts as BFS and DFS to search for the path from the initial state to the goal state. However, UCS needs to compute the cost of each path and use a priority queue to store and sort nodes based on the cost from the root node to that node.

It assumes that all action costs are non-negative. This assumption is important because it ensures that the search algorithm will always find the lowest-cost path to the goal state, rather than getting stuck in a loop of exploring higher-cost paths.

3. Comparing Performance of DFS, BFS, and UCS Algorithms in Sokoban Game

3.1. ***Experimental results***

Based on the experimental results of finding paths using three algorithms DFS, BFS, and UCS on all available Sokoban maps, we can make the following observations:

- **Map 5** is considered the most challenging, it cannot be solved using DFS, while BFS and UCS require a significant amount of time to find a solution.
  - The size of the map in **Map 5** is larger than the other maps in the dataset, leading to a much larger state space that needs to be explored during the search process.
  - DFS exceeded the maxdepth and failed to perform in level 5.
  - UCS finds the optimal solution faster than BFS because it can avoid exploring less promising branches of the search space.
- The path found by DFS is always the longest, while the path found by BFS and UCS is the same and optimal.

Level	**DFS**	**BFS**	**UCS**
1	79	12	12
2	24	9	9
3	403	15	15
4	27	7	7
5	MLE	20	20
6	55	19	19
7	707	21	21
8	323	97	97
9	74	8	8
10	37	33	33
11	36	34	34
12	109	23	23
13	185	31	31
14	865	23	23
15	291	105	105
16	MLE	34	34

Table 1. The number of steps taken by 3 search algorithms DFS, BFS, UCS on all maps.

Level	**DFS**	**BFS**	**UCS**
1	0.05	0.07	0.05
2	0.01	0.01	0.01
3	0.20	0.15	0.10
4	0.00	0.01	0.00
5	MLE	165.44	126.62
6	0.01	0.01	0.01
7	0.46	0.74	0.54
8	0.06	0.17	0.18
9	0.23	0.01	0.01
10	0.01	0.01	0.02
11	0.01	0.01	0.02
12	0.11	0.07	0.08
13	0.16	0.12	0.15
14	3.37	2.44	2.85
15	0.14	0.24	0.25
16	MLE	19.18	18.14

Table 2. The time taken by 3 search algorithms DFS, BFS, UCS on all maps (ms).

3.2. ***Evaluating the solutions found by 3 algorithms***

DFS is fast in finding a solution, but it may not produce an optimal solution as it does not prioritize the shortest or lowest-cost path. While DFS is generally faster than BFS on simple maps, it may run out of memory faster and may be limited by memory constraints on larger and more complex maps.

BFS always explores nodes in the order of their distance from the starting node, making it better than DFS in finding the optimal or shortest path solution for Sokoban. However, BFS may take longer to find a solution because it explores all nodes in each level before moving on to the next level, making it computationally expensive for large search trees. While BFS can reduce the memory overflow compared to DFS by opening nodes in the lowest depth, it can still be more memory-intensive in the long run. The worst-case scenario for BFS is when it reaches the final level of the tree, which can be much slower and worse than DFS.

UCS can explore the search space more intelligently than BFS by using a priority queue to expand the node with the lowest path cost, rather than expanding all nodes at the current depth like BFS. This can lead to a more efficient search in scenarios where the optimal solution has a lower path cost and is located at a deeper level in the search tree. It is possible to optimize the running time of UCS by changing the cost function to reduce computation

In summary, the selection of an algorithm to solve the Sokoban problem depends on the specific needs of the problem. If obtaining the optimal solution is a priority, BFS or UCS would be the preferable options. If the priority is speed and the length of the solution is not a concern, DFS may be more appropriate. **Nevertheless, for solving the Sokoban puzzle, UCS remains the best algorithm of choice.**