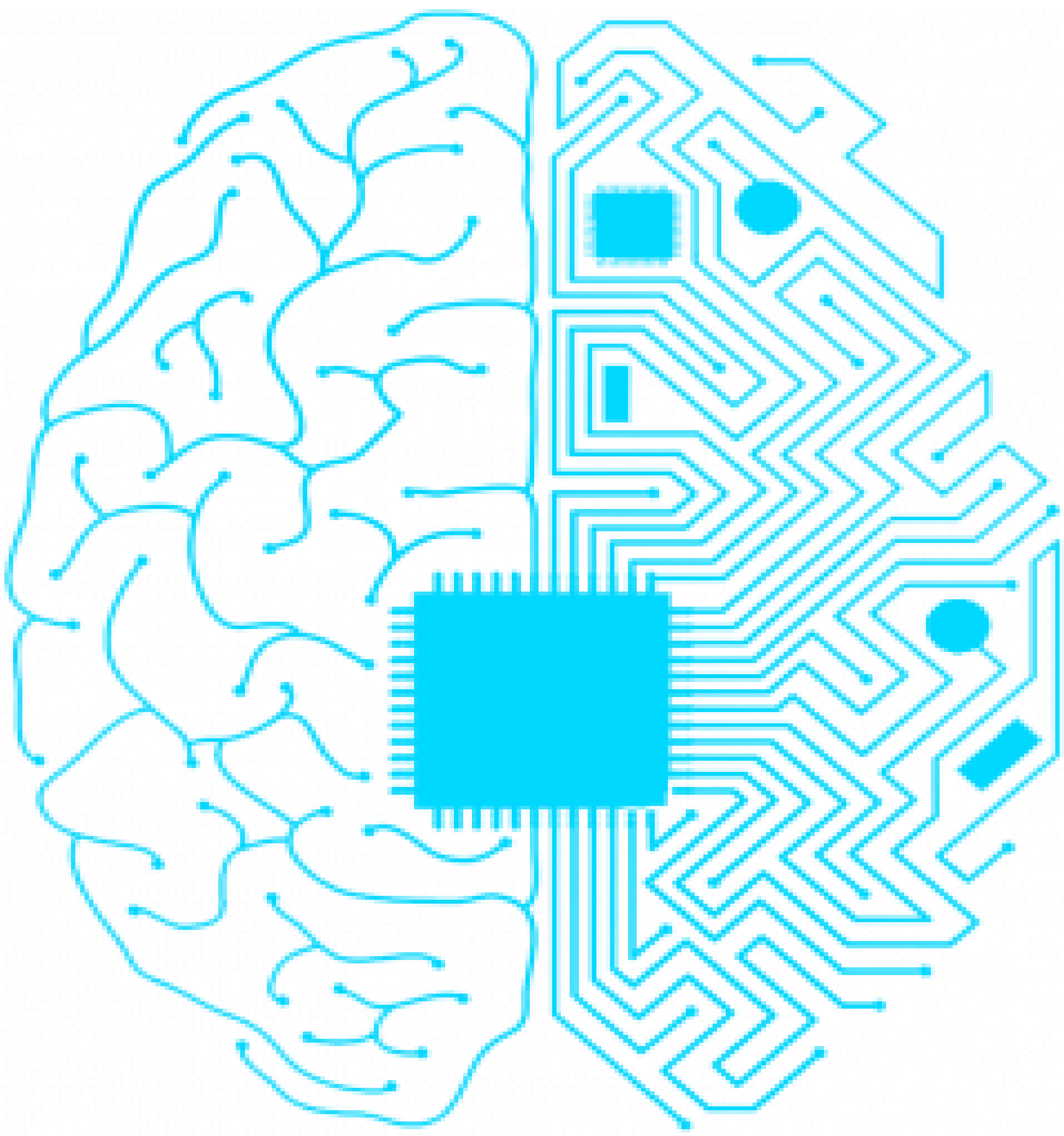

Sokoban – AI1

Head: Dr. Parvin Razzaghi **TA: Ashkan Rahmani Nejad**

Written By: Solmaz Mohamadi - Sepide Bahrami **11.4.2019**

First Phase: Understanding the Problem

For my teammate, and me it was hard to understand how Sokoban game works, because we are not really a big fan of computer games.
Therefore, first, we played it online to figure out how it works and then we started to understand the challenges and difficulties of the game.
The next thing was the programming Language. We chose C++ over Python because the project did not require GUI.

Second Phase: Data Structure

We started to think about what is the best data structure for this game. What is the best data-type to represent our input data?
To answer these questions we have to consider two things:

1. We cannot miss any important data about states of the game.
2. We should use the simplest and smallest data structure to minimize complexity.

After searching and considering our options, we decided to choose a "2D vector" initializing the given characters as "char" which has a size of 1-byte. It is important for us because of the game state being significant; also, it is not possible to create a type less than a byte in C++. (It was also possible to design a specific data-type with smaller capacity, but it would increase runtime of our code)

Third Phase: Initial State

Initial state in this problem can be any matrix representation in which there is an 'S', at least an 'X' and '@', and some '#'s as our blocks.
In this project, we have been given three text documents as inputs of the matrix and our starting states.

Fourth Phase: Goal State

We can define our goal state as follows:
Both 'S' and '@' being in up, down, left or right of 'X' respectively.

```
    S
    @
S @ X @ S
    @
    S
```

Fifth Phase: Successor

We designed and implemented the successor as same as the way it was defined in the classroom. Definition of successor is as follows:

- Successor: A function, which takes a state as an argument and returns a set of states as all of the next possible moves for our agent in Sokoban game.

Therefore, the question is what would be our next moves?
Consider the agent in both of the following states.

#	.	#
.	S	.
#	.	#

First example

In this example state, the agent can just walk up, down, left or right. Therefore, the successor of this state, would give us four new states as answer.

In the next example, you see the agent can push boxes up, down, left or right if the next square of '@' is a dot. In this case, either, the successor would give us four new states as answer.

		.		
	#	@	#	
.	@	S	@	.
	#	@	#	
		.		

Second example

Sixth Phase: Developing Search Algorithms

1. Breadth First Search (BFS)

For developing BFS, we used a pseudocode we learned in class and by using a specific successor, it simply worked. The problem was the execution time for larger dimensions. So we figured if it was a Graph-Based Algorithm, it wouldn't search visited states and it would be much faster. So we added a visited list to save all visited states; and the difference was out-standing.

2. Depth First Search (DFS)

The same as BFS algorithm goes for DFS, considering that DFS is incomplete, it can trap in loops and it may not return answer at all! So it was much more urgent for DFS to be Graph-Based.

3. Iterative Deepening Search (IDS)

For developing this algorithm, we needed two functions. First one named LDS that does a DFS in a specified depth. The second one named IDS, which calls the LDS function iteratively with different bounds until the answer is found in any of the depths. The tree-based version of IDS just returned the answer for the first small input ("Soko1"). However, after using a visited vector to keep the visited states, the execution time decreased impressively and this time we got an answer for a medium sized and large sized sample input, which we tested. Because the examples were easy, we decided to keep all visited states without any limitation.

Comparison of Results

To test and compare the results of the above three algorithms we used three sample inputs as given.
You can see the results in tables below:

Soko1

	Execution time (seconds)	**No. of Pushes**	**No. of Successors**	**No. of Moves**	**Space (kb)**
BFS	0.013	6	93	98	2.24
DFS	0.010	5	83	83	1.8
IDS	0.043	4	47	211	0.8

For the small input size, the execution time of DFS is less than the two others. It shows that for problems with no loops and finite depth, DFS is handier. About space, IDS algorithm is always better than BFS and IDS due to its O(bd) complexity.

Input_2 (13*26)

	Execution time (seconds)	No. of Pushes	No. of Successors	No. of Moves	Space (kb)
BFS	22.57	209	10277	10384	213
DFS	39.01	47	11306	11306	237
IDS	11.19	97	3446	33865	52

For the medium input size, we can see that IDS is pioneer in execution time. In addition, it uses much less space. Then BFS comes in second, which is better than DFS in time complexity. Because DFS goes through the deepest depth.

Input_3 (21*52)

	Execution time (seconds)	No. of Pushes	No. of Successors	No. of Moves	Space (kb)
BFS	0.4	37	894	911	18
DFS	-	-	-	-	-
IDS	0.3	19	352	1656	5.4

For the large input size, IDS and BFS time complexities are almost the same in this example but it can differ in other ones. Again, IDS uses much less space. DFS function did not answer this example though.

<u>Algorithm for Multiple Boxes</u>

_For solving multi-box mode of game, we only need to save number of boxes and number of solved ones; to control the times we run search algorithm for boxes. Although we need to clear successor for next boxes.

Multi Box – Input (5 * 12)

	Execution time (seconds)	No. of Pushes	No. of Successors	No. of Moves	Space (kb)
BFS	0.013 - 135ms	42	674	692	15
DFS	15ms	6	152	152	3.2

Multi Box - Input (8 * 18)

	Execution time (seconds)	**No. of Pushes**	**No. of Successors**	**No. of Moves**	**Space (kb)**
BFS	0.05s	23	358	370	7.7
DFS	4.9s	62	1658	4658	100

As you can see DFS is better for smaller dimensions and BFS is better for larger ones.