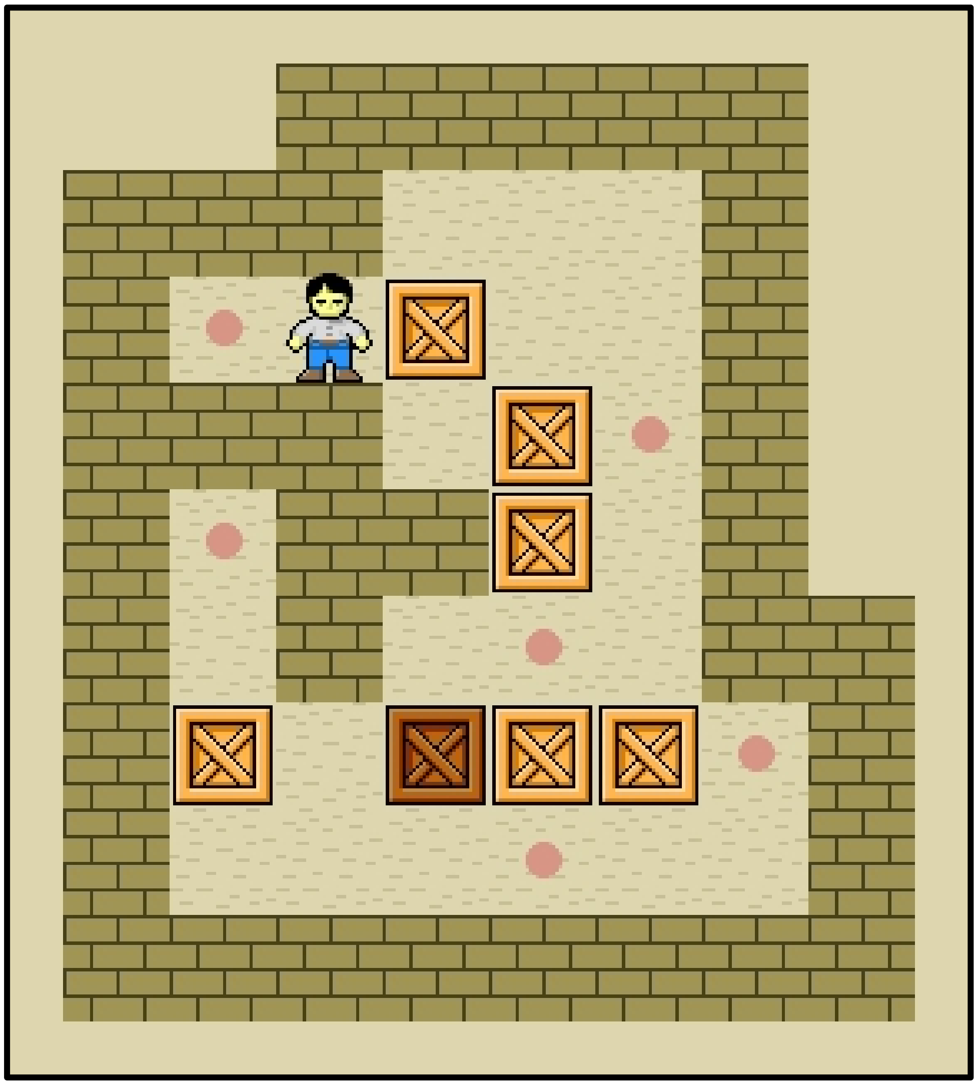
ARTIFICIAL INTELLIGENCE PROJECT



Presented by
Shrey Gupta
2019-IMG-056

Submitted to
Dr. Pinku Ranjan

Title of the Project

Solving **Sokoban** game using Breadth-First search and Depth-First search algorithms and comparing their time and space complexities

Abstract

Sokoban is a popular Japanese Game, we'll be using the most popular uninformed search algorithms to solve this game and further we'll compare the space and time complexities to find out which algorithm is better than the other.

Introduction

Sokoban is a puzzle game in which a warehouse robot must push boxes into storage spaces. The rules hold that only one box can be moved at a time, that boxes can only be pushed by a robot and not pulled, and that neither robots nor boxes can pass through obstacles (walls or other boxes).
In addition, robots cannot push more than one box, i.e., if there are two boxes in a row, the robot cannot push them. The game is over when all the boxes are in their storage spots.

Methodology

Data Structure

For choosing the data structure to be used, we have to consider two things:

- All the important data about states of the game needs to be recorded.
- Data structure which is simple and small would be preferred to minimize complexity of the program.

After searching and considering the options, "2D vector" seems the most appropriate and initializing the given characters is done as "char" which has a size of 1-byte. It is important because of the game state being significant; also, it is not possible to create a type less than a byte in C++.

Initial State

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

Goal State

The goal state can be defined as follows:
Both ‘S’ and ’@’ being in up, down, left or right of ‘X’ respectively.

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

Successor

The successor is designed and implemented similar to 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 ofstates 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 newstates as answer.

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

Second example

Search Algorithms

1. Breadth First Search (BFS)
For developing BFS, a pseudocode is used and by using a specific successor, it simply works. To tackle the problem of the execution time for larger dimensions, a Graph-Based Algorithm is used. It wouldn't search visited states and it would be much faster. A visited list to save all visited states.

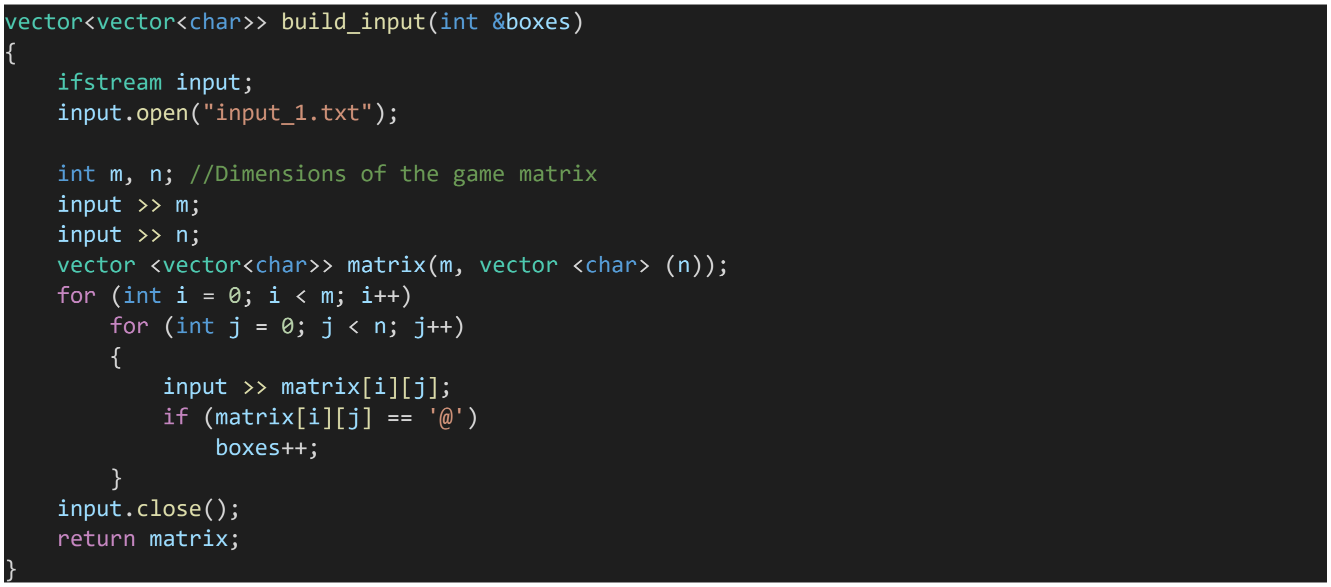
2. Depth First Search (DFS)
The same as BFS algorithm goes for DFS, considering that DFS is incomplete, it can trap in loops and it may not return answer at all. Thus the DFS needs to be Graph-Based.

Driver Code

The entire code for the project is pushed to Github, below I have explained the important functions of the code for Breadth First Search and Depth First Search.

Github Repository: https://github.com/echo-sg/Sokoban

Creating Matrix from Input File:



```
vector<vector<char>> build_input(int &boxes)
{
    ifstream input;
    input.open("input_1.txt");

    int m, n; //Dimensions of the game matrix
    input >> m;
    input >> n;
    vector <vector<char>> matrix(m, vector <char> (n));
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
        {
            input >> matrix[i][j];
            if (matrix[i][j] == '@')
                boxes++;
        }
    input.close();
    return matrix;
}
```

Finding successor of current state on BFS implementation

```cpp
void BFS_successor(vector<vector<char>> state, queue<vector<vector<char>>> &state_q, int& count_push, int& count
_move, int &c_succ, int& bytes)
{
    int x = 0, y = 0;
    for (int i = 0; i < state.size(); i++)
        for (int j = 0; j < state[i].size(); j++)
            if (state[i][j] == 'S')
            {
                x = i;
                y = j;
            }

    if (state[x][y - 1] == '.')
    {
        state_q.push(walk(x, y, x, y - 1, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x][y + 1] == '.')
    {
        state_q.push(walk(x, y, x, y + 1, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x - 1][y] == '.')
    {
        state_q.push(walk(x, y, x - 1, y, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x + 1][y] == '.')
    {
        state_q.push(walk(x, y, x + 1, y, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x][y - 1] == '@')
    {
        if (state[x][y - 2] == '.')
        {
            state_q.push(push(x, y, x, y - 1, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
        else if (state[x][y - 2] == 'X')
        {
            state_q.push(push(x, y, x, y - 1, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);}
    }
    if (state[x][y + 1] == '@')
    {
        if (state[x][y + 2] == '.')
        {
            state_q.push(push(x, y, x, y + 1, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
        else if (state[x][y + 2] == 'X')
        {
            state_q.push(push(x, y, x, y + 1, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
    }
    if (state[x - 1][y] == '@')
    {
        if (state[x - 2][y] == '.')
        {
            state_q.push(push(x, y, x - 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
        else if (state[x - 2][y] == 'X')
        {
            state_q.push(push(x, y, x - 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
    }
    if (state[x + 1][y] == '@')
    {
        if (state[x + 2][y] == '.')
        {
            state_q.push(push(x, y, x + 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
        else if (state[x + 2][y] == 'X')
        {
            state_q.push(push(x, y, x + 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
    }
}
```

Driver Code for BFS algorithm:

```
vector<vector<char>> BFS(vector<vector<char>> &current_state, vector<vector<vector<char>>> &visited_list, int all_boxes, int &finished_boxes, int &c_push, int &c_succ, int &c_move, int& bytes)
{
    queue<vector<vector<char>>> state_q;
    BFS_successor(current_state, state_q,c_push,c_move,c_succ,bytes);

    while (!state_q.empty())
    {
        if (is_goal(current_state))
        {
            finished_boxes++;
            current_state = BFS(current_state, visited_list, all_boxes, finished_boxes, c_push, c_succ, c_move, bytes);
        }
        if (finished_boxes == all_boxes)
        {
            return current_state;
        }
        current_state = state_q.front();
        state_q.pop();
        if (!visited(current_state, visited_list,bytes))
            BFS_successor(current_state, state_q,c_push,c_move,c_succ,bytes);
    }
}
```

Finding successor of current state on DFS implementation

```
void DFS_successor(vector<vector<char>> state, stack<vector<vector<char>>>& state_s, int& count_push, int& count_move, int& c_succ, int& bytes)
{
    int x = 0, y = 0;
    for (int i = 0; i < state.size(); i++)
        for (int j = 0; j < state[i].size(); j++)
            if (state[i][j] == 'S')
            {
                x = i;
                y = j;
            }

    if (state[x][y - 1] == '.')
    {
        state_s.push(walk(x, y, x, y - 1, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x][y + 1] == '.')
    {
        state_s.push(walk(x, y, x, y + 1, state));
        count_move++;
        c_succ++;
        bytes += sizeof(state);
    }
    if (state[x - 1][y] == '.')
    {
```

```
state_s.push(walk(x, y, x - 1, y, state));
       count_move++;
       c_succ++;
       bytes += sizeof(state);
   }
   if (state[x + 1][y] == '.')
   {
       state_s.push(walk(x, y, x + 1, y, state));
       count_move++;
       c_succ++;
       bytes += sizeof(state);
   }
   if (state[x][y - 1] == '@')
   {
       if (state[x][y - 2] == 'X')
       {
           state_s.push(push(x, y, x, y - 1, state));
           count_push++;
           count_move++;
           c_succ++;
           bytes += sizeof(state);
       }
       else if (state[x][y - 2] == '.')
       {
           state_s.push(push(x, y, x, y - 1, state));
           count_push++;
           count_move++;
           c_succ++;
           bytes += sizeof(state);
       }
   }
   if (state[x][y + 1] == '@')
   {
       if (state[x][y + 2] == 'X')
       {
           state_s.push(push(x, y, x, y + 1, state));
           count_push++;
           count_move++;
           c_succ++;
           bytes += sizeof(state);
       }
       else if (state[x][y + 2] == '.')
       {
           state_s.push(push(x, y, x, y + 1, state));
           count_push++;
           count_move++;
           c_succ++;
           bytes += sizeof(state);
       }
   }
   if (state[x - 1][y] == '@')
   {
       if (state[x - 2][y] == 'X')
       {
           state_s.push(push(x, y, x - 1, y, state));
           count_push++;
           count_move++;
           c_succ++;
           bytes += sizeof(state);
       }
```

```
Else if (state[x - 2][y] == '.')
        {
            state_s.push(push(x, y, x - 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
    }
    if (state[x + 1][y] == '@')
    {
        if (state[x + 2][y] == 'X')
        {
            state_s.push(push(x, y, x + 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
        else if (state[x + 2][y] == '.')
        {
            state_s.push(push(x, y, x + 1, y, state));
            count_push++;
            count_move++;
            c_succ++;
            bytes += sizeof(state);
        }
    }
}
```

Driver Code for DFS algorithm:

```
vector<vector<char>> DFS(vector<vector<char>> &current_state, vector<vector<vector<char>>> &visited_list, int &a
ll_boxes, int &finished_boxes,int& c_push,int& c_succ,int& c_move,int& bytes)
{
    stack<vector<vector<char>>> state_stack;
    DFS_successor(current_state, state_stack,c_push,c_move,c_succ,bytes);

    while (!state_stack.empty())
    {
        if (is_goal(current_state))
        {
            finished_boxes++;
            visited_list.push_back(current_state);
            current_state = DFS(current_state, visited_list, all_boxes, finished_boxes, c_push, c_succ, c_move,
bytes);
        }
        if (finished_boxes == all_boxes)
        {
            return current_state;
        }
        current_state = state_stack.top();
        state_stack.pop();
        if (!visited(current_state, visited_list,bytes))
            DFS_successor(current_state, state_stack,c_push,c_move,c_succ,bytes);
    }
    cout << "goal not found" << endl;
    return current_state;}
```

Checking if the goal state is achieved:

```cpp
bool is_goal(vector<vector<char>> &state)
{
    for (int i = 0; i < state.size(); i++)
    {
        for (int j = 0; j < state[i].size(); j++)
        {
            // S->@->X
            if (state[i][j] == 'S')
                if (state[i][j + 1] == '@')
                    if (state[i][j + 2] == 'X')
                    {
                        state[i][j] = '.';
                        state[i][j + 1] = 'S';
                        state[i][j + 2] = 'F';
                        //print_state(state);
                        return true;
                    }
            // X<-@<-S
            if (state[i][j] == 'X')
                if (state[i][j + 1] == '@')
                    if (state[i][j + 2] == 'S')
                    {
                        state[i][j] = 'F';
                        state[i][j + 1] = 'S';
                        state[i][j + 2] = '.';
                        //print_state(state);
                        return true;
                    }
            //top-down
            if (state[i][j] == 'S')
                if (state[i + 1][j] == '@')
                    if (state[i + 2][j] == 'X')
                    {
                        state[i][j] = '.';
                        state[i + 1][j] = 'S';
                        state[i + 2][j] = 'F';
                        //print_state(state);
                        return true;
                    }
            //bottom-up
            if (state[i][j] == 'X')
                if (state[i + 1][j] == '@')
                    if (state[i + 2][j] == 'S')
                    {
                        state[i][j] = 'F';
                        state[i + 1][j] = 'S';
                        state[i + 2][j] = '.';
                        //print_state(state);
                        return true;
                    }
        }
    }
    return false;
}
```

Comparison of Time and Space Complexities

To test and compare the results of the above three algorithms we used three sample inputs as given. The results are as follows:

Input_1

	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

For the small input size, the execution time of DFS is less than BFS. It shows that for problems with no loops and finite depth, DFS is handier. About space, BFS performs better.

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

BFS 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	-	-	-	-	-

BFS was way better than DFS function as DFS did not answer this example.

Results

For smaller input sizes DFS is a better alternative to BFS w.r.t. the time and space associated. However for medium or large input sizes BFS outperforms DFS with smaller execution time and a guarantee of result. For very large input size, DFS simply couldn’t be used as there is no guarantee of solution.

Conclusions

The project helped us in getting an understanding the following points

- How Sokoban is played and the rules associated with it.
- What successor function, goal test and other functions are.
- How using the right search algorithm can drastically effect the overall performance and efficiency.

Resources and References

https://math.stackexchange.com/

https://www.geeksforgeeks.org

https://www.conceptispuzzles.com/

https://towardsdatascience.com/