EE586 Artificial Intelligence Term Project
Final Report
AI Playing Sokoban Game

Group: MasterBots

Başer Kandehir
baser.kandehir@ieee.metu.edu.tr
METU EEE Department

Ferhat Gölbol
ferhat.golbol@ieee.metu.edu.tr
METU EEE Department

Seyit Yiğit Sızlayan
e187686@metu.edu.tr
METU EEE Department

***Abstract*—This report explains the implementation steps of a Sokoban-game-solving artificial intelligence. This project is offered in EE586 Fall semester of 2017 at METU. The project involves various single-agent informed/uninformed search algorithm implementations, their comparisons and applications on a specific GUI.**

I. Introduction

In this project, different algorithms have been implemented to solve Sokoban Game and the results are displayed both with a text-based UI and a web-based GUI. All the details of implementation and results have been explained in this document. Section II gives the definition of the problem, Section III gives the details of the GUI, Section IV explains the algorithmic details and implementation, Section V explains the results, Section VI concludes the report, and finally Section VII gives the work allocation.

II. Definition of the Problem/Task

Sokoban, which means warehouse keeper in Japan, is a transport puzzle game which player tries to put boxes in designated locations by pushing them around a maze. [1]

The rules of the game are as follows:

1. Only one box may be pushed at a time.
2. Boxes cannot be pulled.
3. In a stage of the map, number of desired locations and number of boxes are equal, and it is expected to move these boxes to desired locations by pushing them.
4. As long as all the desired locations are filled, it doesn't matter which box fills which desired location.
5. When all the desired locations are filled, the game ends.
6. A box can be pushed only if the target location is empty, because it is not physically possible to do so in the game.

Figure 1 shows an example stage from Sokoban Game. The green ball represents the player, white squares represent the desired locations, and purple squares represent the movable boxes. Brown squares represent the walls, and the all other squares are empty locations.

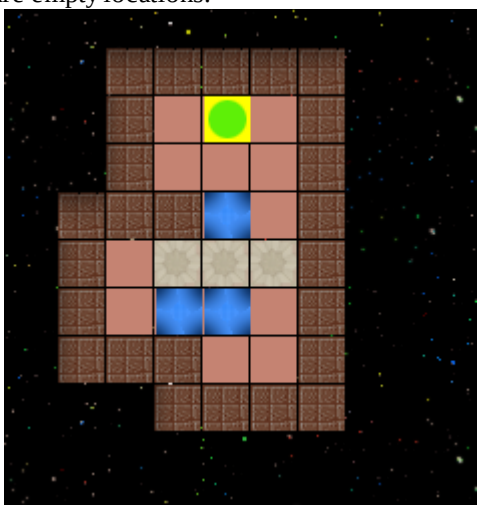

Figure 1 Example Stage from Sokoban Game

III. GUI

A. Text-Based UI

The main solver program, which runs the search algorithms given in Section IV, is implemented in C++. The executable takes the path to the puzzle and the desired search algorithm as command line arguments and writes the generated solution in "results.txt". A sample run of the program is "ee586.exe puzzle.txt y".

Input file is a plain text file. The first line contains M, N, the number of rows and columns as two space separated integers. Each of the following M lines contain exactly N ascii characters from the following list:

@	: the player
space	: empty square
$	: movable box
#	: wall

. : desired location
+ : the player at a desired location
* : box at a desired location

The search algorithm is one of the following options:

e : BFS
t : IDDFS
y : A* with Deadlock Checking
u : A* with Manhattan Distance
i : A* with Euclidean Distance
o : A* with Deadlock + Manhattan Distance
p : Deadlock Checking BFS

The output file is also a plain text file. The first line contains T, the time elapsed in microseconds. The second line is D, the solution depth. Each of the following D lines contain an action, u/d/l/r, corresponding to upward, downward, left and right movements respectively.

B. Web-Based GUI

There are lots of GUI implementation of sokoban game on the web. Some of them is offline, some of them are online. For the project, since they have player score database, we have preferred the GUI implemented on www.game-sokoban.com. The GUI is simple: Green/yellow box is player and it is moved using arrow keys, blue squares are the boxes and grey squares are the desired locations. When a box is on one of the desired locations a little green arrow appears at the corner.

To be able to implement it on our problem, we decided to convert this web-based GUI into the text-based problem so that the solver we have implemented can work on it. Then obtaining the solution, the program will take the solutions and apply them on the web page. The steps are as follows:

1) Extracting Map Data from the HTML Page

On the web page, the map is in a simple HTML file and the game rules are applied using a JavaScript code. After a little inspection, we have noticed that all the maps are on 20X20 HTML table. The background of each cell is changed according to the map if the cell is a "wall" or "desired location".

The boxes and the player are as "div"s on the HTML page and by changing their location values they are moved.

We have used Python as a language and Selenium Web Driver and BeautifulSoup libraries, which is mostly used for analyzing web page behaviors, to open the web page, inspect it then convert it to a text-based map. The algorithm is as follows:

This algorithm returns a character matrix on which all the walls and desired locations are signed.

- Open the web page using selenium web driver.
- Obtain the HTML source from it.
- Find the table whose ID is "th4". The game map is implemented in this table.
- Separate each cell to form cell matrix.
- For each line:
    - If line contains no wall or desired location cell
        - Delete the line
- Take the transpose of the matrix
- For each line:
    - If line contains no wall or desired location cell
        - Delete the line
- Take the transpose again to obtain final cell map
- Form an empty character matrix as the same shape as final matrix
- For each cell
    - If cell background is "Wall"
        - Make corresponding character in character matrix "#"
    - If cell background is "Desired location
        - Make corresponding character in character matrix "."

Next task is finding the indexes of the player and boxes on the map matrix. The algorithm is as follows:

- Find cell dimensions from html
- Find position of table "tb4"
- Find position of division "mv1" which is position of player
- Find positions of the divisions whose classes are "bx1" which are box list
- The indexes of player:
    - Player.Y = (Mv1.y - Tb4.y)/(cell_dimension +1)
    - Player.X = (Mv1.x - Tb4.x)/(cell_dimension +1)
- For each i^{th} division in bx1 list:
    - Box_i.Y = (Bx1[i].y - Tb4.y)/(cell_dimension +1)
    - Box_i.X = (Bx1[i].x - Tb4.x)/(cell_dimension +1)
- For each character in map matrix:
    - If cell is desired location and player position
        - Change it to "+"
    - Else if cell is player position
        - Change it to "@"
    - Else if the cell is desired location and there is a box
        - Change it to "*"
    - Else if there is a box
        - Change it to "$"
    - Else
        - Do not change
- Save character matrix to "sokobanGame.txt"

This algorithm returns a character matrix which is the text-based version of the map on the web-page.

2) Calling the Solver and Obtaining the Solution

Finding the text-based version of the map, to solve the map the solver program written in C++ is called with wanted algorithm inside Python program. The Python program is blocked until the solver returns and writes the solution to "results.txt". Then the Python program reads that file.

3) Applying the Solution to Web-Based GUI

When the file is read, the solution is ready to apply. The solution is applied to the web page as arrow keys. The sequential keys are pressed with 1s period using Selenium library.

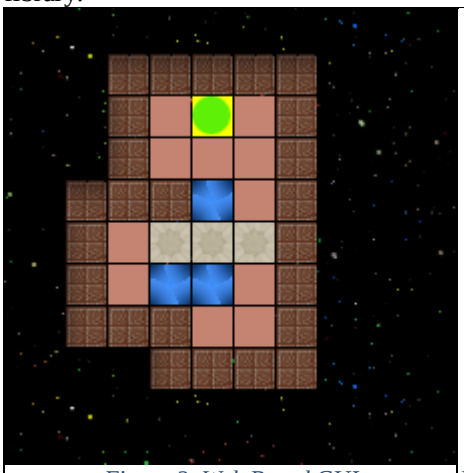

Figure 2: Web Based GUI

```
8 6
 #####
 # @ #
 #   #
###$ #
# ...#
# $$ #
###  #
  ####
```

Figure 3: Converted Text-Based GUI, problem size at the top

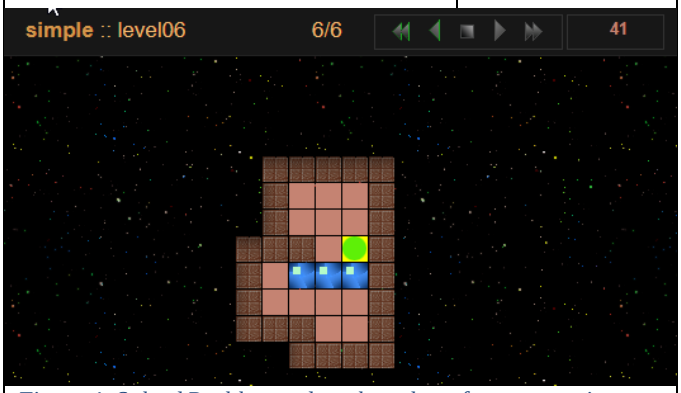

Figure 4: Solved Problem and total number of movements is at up-right corner

IV. Algorithmic Details and Implementation

To solve the Sokoban Game, we have used Breadth First Search, Iterative Deepening Depth First Search and A* algorithm with different heuristics.

We haven't used Depth First Search algorithm for solving this game, because it wasn't suitable for solving this game for various reasons. For example, if DFS chooses to go left when it should have gone to right, it exhaustively searches for a solution in the wrong place. This takes too much time and the algorithm may even enter an infinite loop. Instead of using DFS, we have used IDDFS which limits the depth of DFS and increases the depth gradually until a solution is found.

BFS is generally a much better option compared to DFS when solving games like Sokoban because it doesn't enter loops like DFS does and it is generally much faster.

A* is even better than BFS and IDDFS, because instead of blindly searching for the solution, it uses an heuristic that helps it find the solution much faster.

A. Breadth First Search (BFS)

- V_0 := {initial state}
- previous[initial state] := NULL
- k := 0
- while not in target state and Vk is not empty repeat
  - V_{k+1} = {}
  - foreach node s in V_k
    - foreach successor sp of s
      - if sp is not visited
        - visited[sp] := true
        - prev[sp] = s
        - push sp into V_{k+1}
  - k := k+1
- construct the solution using prev[] map (backpointers)

BFS is implemented in this project in two different forms: BFS as studied in the class, whose pseudocode is given below, and deadlock checking BFS. The latter does not push the successor puzzle in the queue if a deadlock exists. The rationale behind this action is, if a box is pushed to a corner which is not a desired location, it cannot be pulled back, thus no solution can result from this puzzle.

B. Iterative Deepening Depth First Search (IDDFS)

```
bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit?1)
            return true
    return false

// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
       if DLS(src, target, limit) == true
           return true
    return false
```

C. A-Star Algorithm and Heuristic Functions

```
• f[initial state] := 0
• prev[initial state] := NULL
• PQ := {initial state}
• while PQ is not empty repeat
  o pop node s from PQ
  o if s is target state return
    SUCCESS
  o mark s as visited
  o foreach successor sp of s
    ▪ if sp is not visited
      • prev[sp] = s /* set
        backpointer */
      • f[sp] = f[s] + 1 /* depth
        */
      • push sp into PQ with
        priority f[sp] + h[sp],
        where h is heuristic score
```

1) Deadlock Checking

When a box is pushed to a non-desired corner, no solution can be obtained from its children. This is indicated by an infinite heuristic value. Thus, the states with a deadlock are pushed to the end of the priority queue, and never expanded provided that a solution exists.

2) Manhattan Distance

In this part, sum of the Manhattan distances of each box to the nearest desired location is used as heuristic. This heuristic is covered and shown to be admissible in [2].

3) Euclidean Distance

Instead of the Manhattan distances, sum of the Euclidean distances is used as heuristic. Since the Euclidean distance is smaller than or equal to the Manhattan distance, this heuristic is also admissible.

4) Deadlock Checking and Manhattan Distance

As explained in Russell and Norwig, the maximum of two or more admissible heuristics is also admissible. Since deadlock check returns zero if a solution might exist and infinity otherwise, sum of deadlock check and another heuristic is the maximum of the two, which is admissible.

V. Results

We run the algorithms BFS, IDDFS, A* with 4 different heuristics on 170 different levels with optimal step sizes changing from 10 to 178, and obtained the results of our implementation. We will compare and comment on these results in this section.

Figure 5 shows the results of BFS algorithm. X axis shows the step of size of the levels, and y axis shows the execution time in microseconds. We had different levels with same step sizes so we calculated the mean, max. and min. execution time of each of those step sizes. Red line represents the mean, blue dots represent the min and yellow dots represent the max. for each of the step sizes. If we were to comment on the results of the BFS algorithm, we can say that as the step size of the problem increases, execution time also generally increases. In other cases, there are some other factors affecting the execution time other than the step sizes, that causes to take more time even when the step size is relatively small. Most obvious factor is the branching factor. As branching factor increases for a problem with same step size, execution time also increases. After we solved all the given Sokoban problems, we have also calculated the max. time it takes to solve a problem with BFS. It took approx. 52 seconds to solve a Sokoban problem in the worst case.

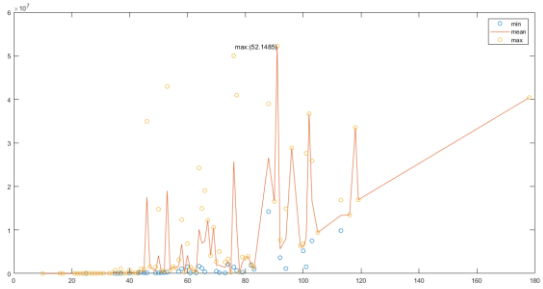

Figure 5 Results of BFS Algorithm

Figure 6 shows the results of IDDFS algorithm up to levels with optimal step size of 39. IDDFS took 2147 sec. to solve the last problem, and was taking a lot more time in the next problem, so we stopped the execution. We actually expected IDDFS to take a lot of time in some of the problems. Because it uses a depth limited DFS and gradually increases the depth. In some problems, it causes the algorithm to exhaustively look for end of the branches and this takes more time as the depth limit increases. Therefore, we can say that IDDFS is not suitable for solving Sokoban problems. In most of the other problems, IDDFS is comparable to BFS in terms of execution time. Figure 7 compares and IDDFS and BFS. Worst case execution times of BFS are much better than those of IDDFS and most of the time BFS performs better than IDDFS in Sokoban game.

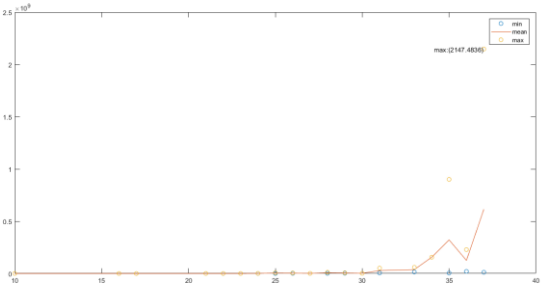

Figure 6 Results of IDDFS Algorithm

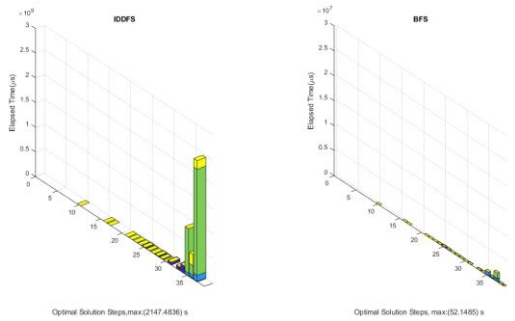

Figure 7 IDDFS vs. BFS

Figure 10 shows the results for A* algorithm with 4 different heuristics which are Manhattan Distance, Euclidean Distance, Deadlock Only, Deadlock combined with Manhattan heuristics. Also results of Deadlock checking with BFS is included in the Figure 10. When we compare the resulting graphs, Deadlock combined with Manhattan heuristic performs the best among them. Manhattan Distance and Euclidean distance heuristics performs very similar. Deadlock heuristic performs much better than Manhattan heuristic when they are used separately. Max. time it takes for each algorithm is written below the graph of each algorithm.

Figure 8 compares BFS with A* with Deadlock and Manhattan Heuristic. As can be seen from the graph, A* performs much better than BFS. Worst case time for BFS is 52.15 seconds whereas it is 3.54 seconds for A* algorithm. This was expected since A* searches intelligently instead of a blind search.

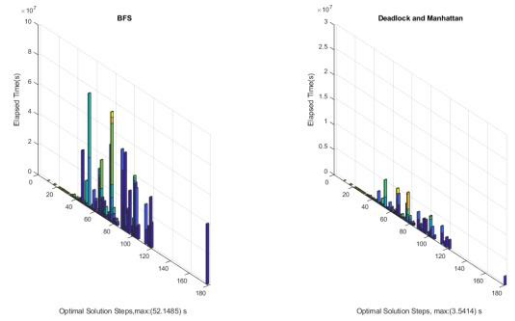

Figure 8 BFS vs. A with Deadlock and Manhattan Heuristic*

We have also added deadlock checking to our BFS algorithm and compared it with the previous BFS algorithm. Figure 9 shows the comparison results. The worst-case time for BFS was 52.15 seconds, and the worst-case time for BFS with deadlock is 8.92 seconds. Therefore, we can say that deadlock checking significantly improved our BFS algorithm.

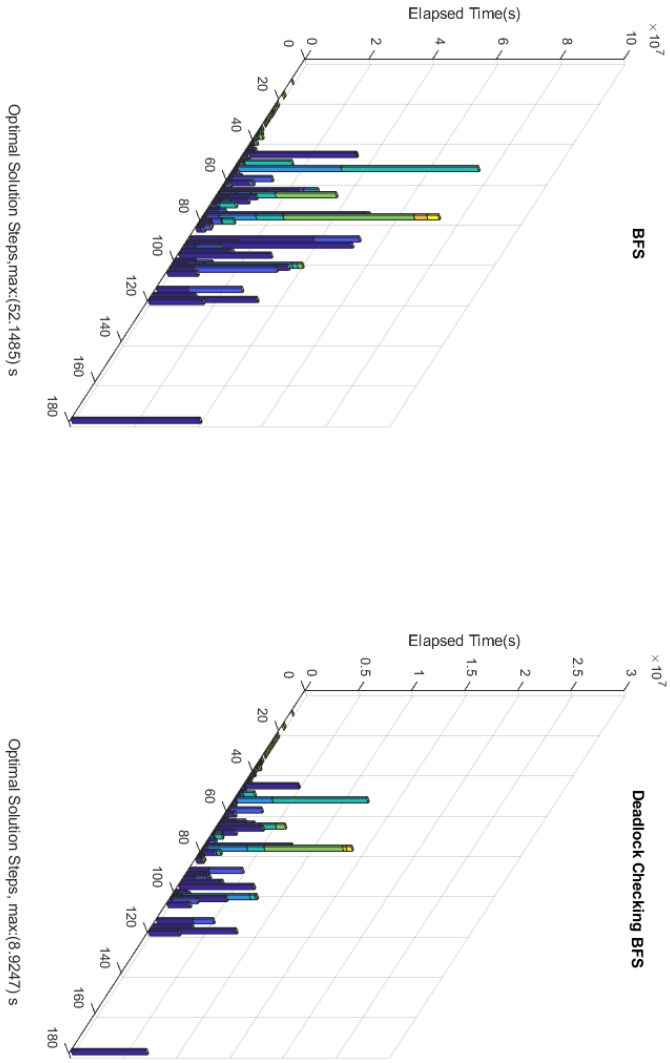

Figure 9 BFS vs. BFS with Deadlock Checking

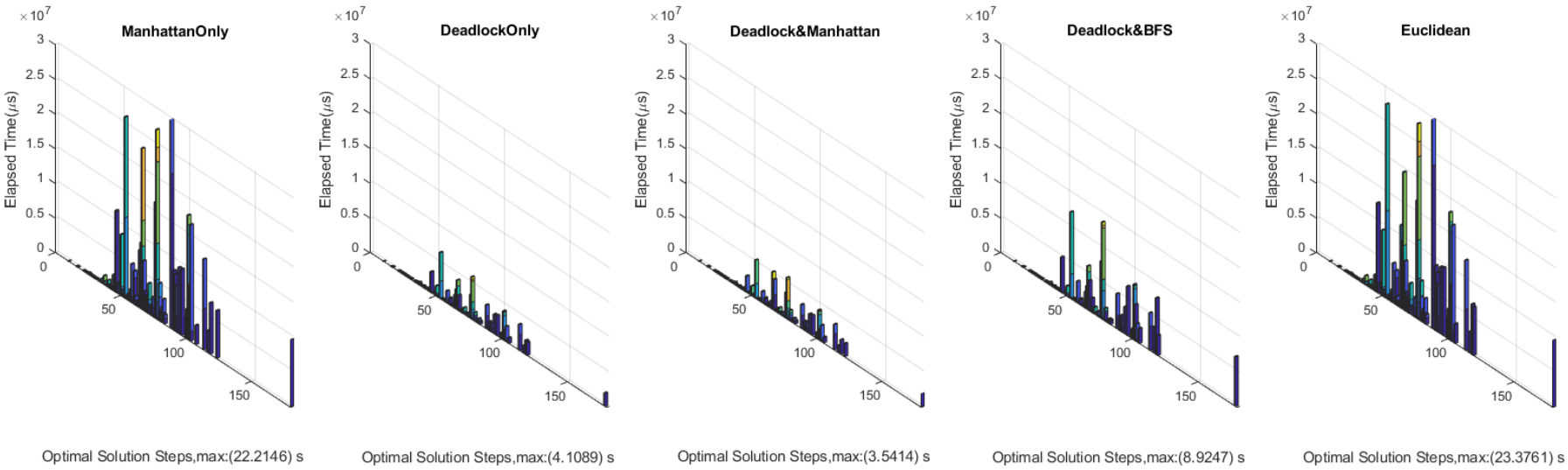

Figure 10 Results of A Algorithm*

VI. Conclusion

In this project, we have solved the Sokoban Game using various algorithms and various heuristic functions. Working on this project allowed us to practice our C++, Python and MATLAB skills on an entertaining game. We have learned how to model a search problem from a given problem definition. Although we have solved several problems with our solver, still in moderately large maps which require lots of steps to solve, it takes too much time to find a solution. In larger maps, we have encountered memory allocation problems. To solve the memory problem, iterative deepening A* algorithm may be used as a future research. Also, there may be better heuristic functions which may result in less number of nodes expanded.
All in all, it was an instructive experience and we have developed a good insight on how solve AI problems.

VII. Work Allocation

Ferhat Gölbol

- Map class implementation
- BFS implementation
- Heuristics

Başer Kandehir

- IDDFS implementation
- Deadlock Checking BFS
- Extracting Map Data from the HTML Page

Seyit Yiğit Sızlayan

- A* implementation
- Solving the Obtained Map and Playing the Game

VIII. References

[1] G. Tamolyvnas, "Sokoban / About," 2012. [Online]. Available: http://www.game-sokoban.com/index.php?mode=about. [Accessed 20 12 2017].

[2] A. Junghanns, *Pushing the Limits: New Developments in Single-Agent Search*, Edmonton, 1999.

[3] G. Tamolyvnas, "Game Sokoban," 2012. [Online]. Available: http://www.game-sokoban.com/index.php. [Accessed 20 12 2017].

[4] Z. Li, L. O'Brien, S. Flint and R. Sankaranarayana, "Object-Oriented Sokoban Solver: A Serious Game Project for OOAD and AI Education," in *Proceedings of the 44th Annual Frontiers in Education Conference*, Madrid, 2014.

[5] M. Suleman, F. H. Syed, T. Syed, S. Arfeen, S. Behlim and B. Mirza, "Generation of Sokoban Stages using Recurrent Neural Networks," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, pp. 466-470, 2017.

[6] M. Abel, 2008. [Online]. Available: http://www.abelmartin.com/rj/sokobanJS/colecciones-txt/pufiban.zip. [Accessed 19 12 2017].

[7] G. Tamolyvnas, "Game Sokoban," 2012. [Online]. Available: http://www.game-sokoban.com/index.php?mode=level&lid=7&version=4. [Accessed 20 12 2017].