# Pushing Boxes with A Mobile Robot in Sokoban-like Scenarios

Uriel H. Hernandez Belmont, Victor Ayala-Ramirez and Raul E. Sanchez-Yanez
*Universidad de Guanajuato DICIS*
*Carr. Salamanca-Valle Km. 3.5+1.8, Palo Blanco*
*36700 Salamanca, Mexico*
*hailehb@laviria.org, {ayalav, sanchezy}@salamanca.ugto.mx*

## Abstract

*This work presents a system to implement the solution of a box-pushing problem inspired in the well-known Sokoban game using a mobile robot. We present in this paper the strategy to determine a sequence of actions that have to be performed by the mobile robot to accommodate a number of boxes in a scenario only by pushing them. We link this solution to actual motion primitives of the mobile robot by decomposing the action sequence. We discuss how the solution is searched and several issues that arise during the search procedure.*

## 1. Introduction

Mobile robotics is an interdisciplinary research field where the main goal is to develop autonomous systems. Some of the components of this autonomous behavior include: to understand the scenario where the robot evolves, to make decisions in an intelligent way, etc. Autonomy will enable us to assign more complex tasks to the robots.

Some examples of complex tasks to be executed by mobile robots could be: to handle different objects with skill [9], to explore planets in space missions[1], to drive automatically unmanned transportation system, etc[11].

The main elements embodied on a mobile robot are:

**actuators,** that enable the robot to modify the scenario where it evolves,

**sensors,** that are useful for enabling the robot to perceive relevant objects and events in its work space,

**processing units,** that are used for the robot to store knowledge about its capabilities; and to make decisions that link perception to action.

As we can see, mobile robots may have many different capabilities. However, tasks can be difficult to accomplish if there exist restrictions in the actuation skills of the robot. In this paper, we consider such a situation, we try to use a real mobile robot to solve some Sokoban-like scenarios.

The game was created by Hiroyuki Imabayashi in 1981. The game consists of a player that needs to push boxes from a starting configuration to a goal setup. The puzzle game is completed when we have attained the goal configuration. We show in Figure 1 an example of a Sokoban board.

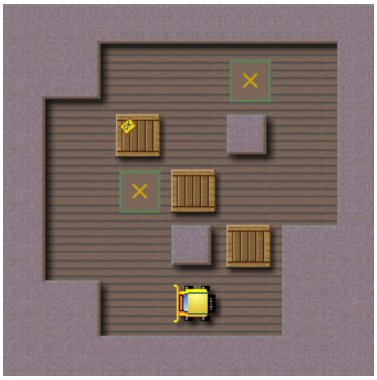


**Figure 1. A typical Sokoban board where the main elements of the problem are displayed.**

The difficulty of the game arises from the constraints of actuation and displacement of the player. We will reproduce this scenario but considering a mobile robot as the player. Boxes can only be pushed and the player can only move to the four-neighbor positions adjacent to its current position.

The complexity of solving the Sokoban game has shown to be NP-hard. Some works dealing with this issue are [5],[6]. In robotics, there have been also some efforts that concern with box pushing [7], [3].

The problem become interesting because the constraints that it imposes to the robot. We will address this problem and we will show how some strategies can help to solve it for a numer of complex scenarios. We use a mobile robot as the agent.

The robot needs a number of elements to solve the problem: knowledge about the environment, knowledge about its capabilities for executing tasks and a decision making system. The knowledge about the environment is represented in a map. The robot needs to be able to execute displacements in a sequential manner. The robot needs also to plan what is the action sequence that will complete its task.

## 2. Methodology

### 2.1. Problem formulation.

The Sokoban game is played by only one agent. The agent must push a number of boxes to an equal number of goal positions. The agent has no pulling capabilities. Some problems arise in the automatic solving of this task. First of them is associated to the representation of the current state of the board, that is we need some kind of map. This map serves to store information about what kind of elements occupy the grid positions of the map.

The second problem is associated to path planning of the agent. The robot needs to move from its current position to a position adjacent to one of the boxes in order to be capable of pushing it. The robot needs to be capable of determining if there exists a series of actions that can attain the goal state. If so, the robot needs to decompose this action set in elementary actuation steps. These elementary actuation steps will consist of simple motion commands or of motion and push commands. The planning process is executed on the map used to represent the environment. In our case, the map used is an occupancy grid where the Sokoban game elements are represented.

### 2.2. Description of the Sokoban elements

The elements that are relevant for our problem are described below:

**The boxes,** are the objects that we need to push to a goal position. We do not known in advance what is the goal position that corresponds to a given box. The boxes, as the player, can only be pushed in north, south, east and west adjacent positions of its current position.

**The agent,** or player, is in charge of pushing boxes in such a way to attain a goal configuration of the entire scenario. Player can only attain 4-neighbor adjacent positions from its current position. If the target position of its movement contains a box, this one is pushed in the direction of the motion of the agent. If such motion is bocked by a wall or by another box, the robot can not move in that direction.

**The goals,** are the set of positions that need to be occupied by boxes in the final state of the scenario.

**The environment** is the occupancy grid representation of the scenario that includes all the elements described before.

Each grid in the map can only have seven values:

**Free:** when there is no element on it.

**Agent:** when the robot occupies that cell.

**Agent on Goal:** when the agent is on a goal cell.

**Box:** when there is a box in that position.

**BoxOnGoal:** when a box is in a goal position.

**Goal:** when there is no other element on a goal

**Wall:** when a wall limiting the environment occupies that cell.

We classify cells with respect to the value changes that can suffer into two types:

**Fixed State Cells (FSC):** when the state of that cell will not change along the problem solution, and

**Dynamic State Cells (DSC):** when the the state of the cell can change during the problem solving process.

DSC can have two or more values during the process of solution. For example, if the initial state of a DSC is Free, the cell can also show the Agent or Box values during the solving procedure.

### 2.3. Path planning

An essential module for solving the Sokoban problem using a mobile robot is the path planner. The Sokoban problem solver will determine what are the starting point and the final position of a robot motion during the solving process. However, the path planner will determine what is the optimal motion plan that the robot needs to execute in order to attain its target position.

The A* algorithm provides an efficient solution to the path finding problem in occupancy grid maps. This algorithm give us the optimal path between two cells on the grid. The A* algorithm has been previously used both in mobile robotics and in video games applications. One important feature of the A* algorithm is that finds the path if here exists one.

As we have said before, the A* algorithm represents the scenario using an occupancy grid [10], [4]). Figure 2 shows an example of such a grid. On it, a white square corresponds to free space and a black square represents an obstacle. The set of empty cells is called the search area, because it is there where the A* algorithm performs its search.

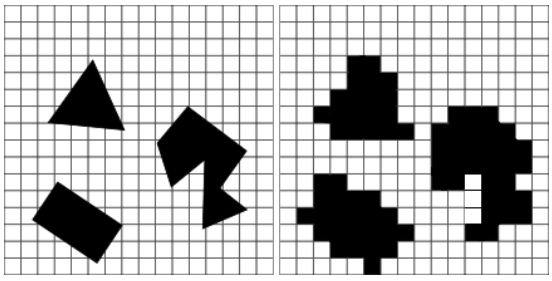



**Figure 2. (a) A typical scenario including obstacles, (b) Its corresponding discretized occupancy grid.**

The second element of the A* algorithm is the cost function. This function will be used to evaluate the cost function and it is expressed as $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of moving the agent to adjacent positions from the current position. Function $h(n)$ is the cost from the current node position to the goal. In the case of path planning, the cost function $h(n)$ is generally taken as the Euclidean distance to the goal from the current position. In this paper, we have made the same choice.

The search process starts from the evaluation of the adjacent nodes of the initial node. We use the adjacency notion of the 4-neighbors because the constraints imposed by the problem. All the nodes adjacent to the current node are added to what is called an *open list*, and they are sorted according to its cost. Simultaneously, we maintain another list, called the *closed list* where we store all the nodes that have been already analyzed, to avoid further processing of them.

We need also to store a parent relationship to know from which node we have evaluated a node entering to the *open list*. The parent node is recorded for all the adjacent nodes evaluated from them. In this way, we can backtrack the entire path by following the parent nodes up to the initial node.

At each node, we choose the next node by selecting the node with the lower cost in the *open list*. If the next node is not the target node, we repeat the above procedure. Otherwise we have solved the path planning problem. If we run out of nodes to analyze without attaining the goal node, the problem has no solution.

## 3. Our approach.

Our approach to solve the Sokoban problem consists of the exploration of a tree where the nodes are the configurations generated from the initial configuration. The search ends when the goal configuration is attained. That is, when all the goal positions are occupied by boxes. The result of this procedure is a list of subtasks that the robot has to execute to solve the problem. As a result of the execution of each sub-task, the scenario changes into a new state. The responsible to command the robot to provoke these changes is the path planner module. The path planner has to translate the subtask into a script of motion commands for the robot. An execution controller has to verify that the robot actually moves accordingly in order to complete task.

We will firstly present how the configuration search tree is generated. We will present then the main issues associated to this process. Afterwards, we will discuss the search procedure.

### 3.1. Configuration generation

To create a new configuration from a given configuration, we perform the following steps.

- We verify the deadlock condition of the node.
- We evaluate the number of liberties for each of the boxes in the environment.
- We search for a path from the agent current position to each of the positions occupying liberties of all boxes in the scenario.
- If there exist some paths from each new position, we repeat the procedure.

The above procedure serves to avoid generating new configurations where the only change is the motion of the robot. We explain more in detail the deadlock condition and the number of liberties of a box in the following paragraphs.

### 3.2. Deadlock

A deadlock configuration is a configuration where there are no possible motion for any box in the environment. Such configurations make impossible to solve the problem. An example of such a configuration is shown in Figure 3 where the box is blocked on the left by a wall and the only liberty is to move aside the wall, but there is no goal in the attainable positions. From such configuration is impossible to reach to goal state.

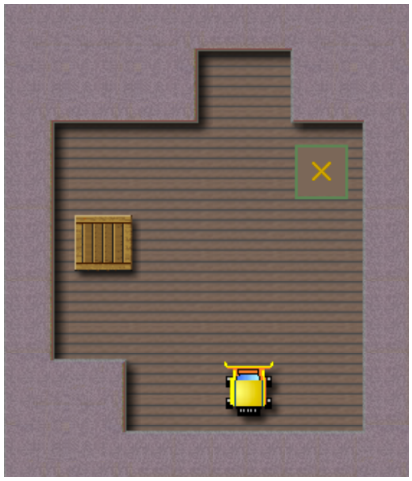


**Figure 3. Diagram showing a deadlock configuration example.**

### 3.3. Number of liberties.

The number of liberties is the number of configurations that a box can attain when pushed by the agent. Figure 4 shows an example of the evaluation of the number of liberties for a Sokoban board. We can see there that two boxes have only two liberties whilst the third one has four liberties. When two liberties of different boxes attain the same position in the grid, we say that those boxes share that liberty.

The number of liberties is independent of the position of the agent, they depend only of the boxes positions. The knowledge

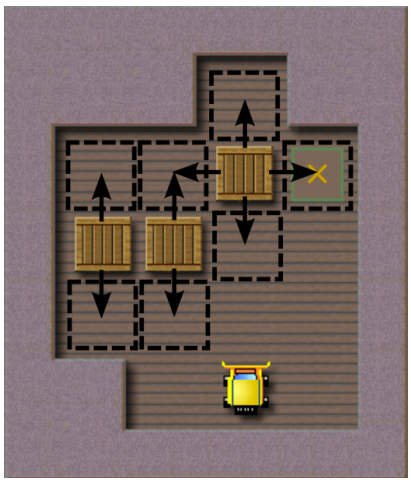


**Figure 4. Diagram showing how liberties are defined(inspired from the Go game nomenclature).**

about the liberties of the boxes in the environment is exploited by the path planner. The goal position for a path has to be a liberty position. for example, in Figure 4, all the boxes can be pushed vertically (form North to South) but the agent can not attain such positions to perform the pushing. As a result, these liberties are discarded.

### 3.4. Solution Search.

In order to find the solution, we explore the configuration tree. Each node of the tree has child nodes. Each child node has a parent node. The search procedure is stopped when the solution node (representing the goal configuration) is attained. We backtrack the parent nodes of all the intermediate nodes in order to know the sequence of configurations that has to be followed to complete the task.

For searching the tree we use a stack and the procedure to the search is as follows:

1. To evaluate the initial node and to set it as the root node.
2. To generate new configurations from this node, setting the parent/child relationships for all nodes.
3. To push the node into the stack.
4. To pop a node form the stack and check if the node is:

   **A solution:** In this case we, look for the root node, returning the back tracking path. We finished the search process.

   **A deadlock:** In this case, we return to step 4.

   Otherwise,

5. We generate new configurations from the node under analysis, setting the parent/child relationships for all the new nodes.
6. Return to step 4.

If we explore the entire configuration tree without finding a solution node, we return a no solution message.

## 4. Test and Results.

We have implemented our approach using C language. We have used the format XSB to represent the maps of the sokoban problems posed by Murase Yoshio [8].

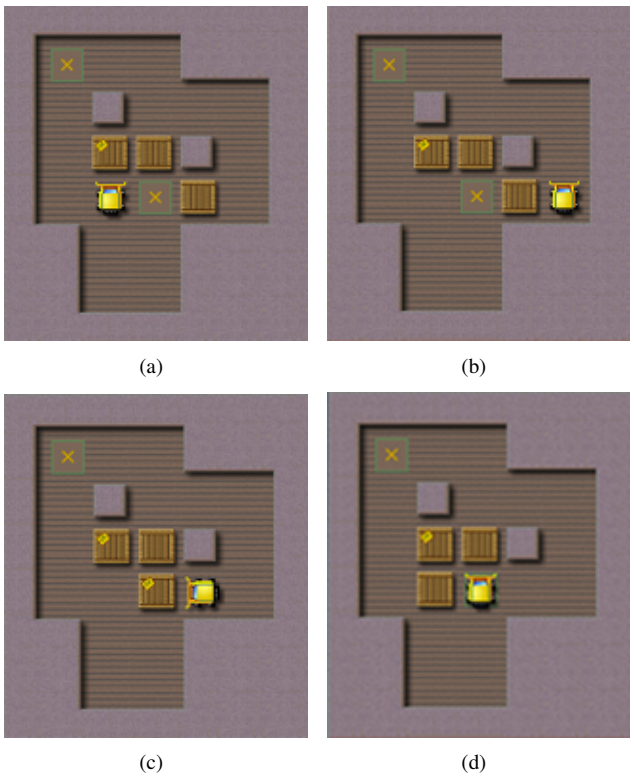
The main goal of the tests in this section is to show an example of a non-trivial board where our approach solves successfully the problem. We present some issues that reveal some important points of the problem.

The test case under analysis is shown in Figure 5. Here, the robot needs to push boxes that are already placed in goal positions to be able to accomplish the task. From the configuration shown, the robot will generate two new configurations: the first one, by pushing the box in front of it in the North direction: and the second one by pushing the second over it. There are no other liberties to generate new configurations. Our system has shown to be affective to solve configurations like the presented in Figure 5.

**Figure 5. The Sokoban board of problem No. 10 from Murase's database [8].**

The solution provided by our system is exemplified in Figure 6 (a)-(d). There are two cases that are interesting to analyze with respect to the generation of motion command for the mobile robot. The first case to observe is when the robot changes of box being pushed (Figures 6 (a)-(b)). In such a case, the A* path planner has to return a motion script to attain the required position for the robot. The second case in when the robot needs to push boxes. In the transitions from Figure 6 (b) to Figure 6(d), we see that a box is pushed two positions in the same direction. We will need to integrate such motions in a single motion command to execute pushing in a given direction.

As we can see from this example, the implemented system returns sub-goal configurations from the current position of the



(a) (b) (c) (d)

**Figure 6. Some steps in the Sokoban problem solving procedure implemented in this work.**

robot. From these configurations, we obtain the motion primitives that the robot needs to execute. This execution is verified to ensure that the robot has accomplished that sub-goal.

In our implementation, we return the first sequence of configurations that attains the goal configuration. We can not ensure that it is optimal with respect to the number of movements. Nevertheless, the objective of analyzing what are the main issues for implementing Sokoban solution in mobile robots has been fulfilled

## 5. Conclusions and future work.

We have presented a method to solve Sokoban problems using mobile robots. We have discussed main details of our approach. We have also presented a test case to validate our approach.

We have shown the importance of decomposing the solution into a sequence of partial goals that are highly correlated to mobile robotics problems. In particular, we have addressed task planning and path planning problems. There is also a component of execution control that has to be addressed in the implementation of this approach on a robotics experimentation platform.

Future work will be directed towards optimizing the search procedure in the Sokoban problem, specifically to improve deadlock detection; and to improve implementation issues in a mobile robot platform due to object interaction and real time execution and completion of the Sokoban task.

## acknowledgements

Uriel H. Hernandez-Belmonte gratefully acknowledges Mexico's CONACYT for the financial support through the scholarship 329356/229784.

## References

[1] Max Bajracharya, Mark W. Maimone, and Daniel M. Helmick. Autonomy for mars rovers: Past, present, and future. *IEEE Computer*, 41(12):44–50, 2008.

[2] David M. Bourg and Glenn Seemann. *AI for Game Developers*. O'Reilly, 2004.

[3] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Comput. Geom.*, 13(4):215–228, 1999.

[4] Greogory Dudek and Michael Jenkin. *Computational Principles of Mobile Robots*. Cambridge University Press, 2010.

[5] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Canadian Conference on AI*, pages 1–15, 1998.

[6] Andreas Junghanns and Jonathan Schaeffer. Sokoban: improving the search with relevance cuts. *Theor. Comput. Sci.*, 252(1-2):151–175, 2001.

[7] M.J. Mataric, M. Nilsson, and K.T. Simsarin. Cooperative multi-robot box-pushing. volume 3, pages 556 –561 vol.3, aug. 1995.

[8] Yoshio Murase. 52 maps generated automatically. `http://www.ne.jp/asahi/ai/yoshio/sokoban/auto52/index.html`. [Online; accessed 19-July-2010].

[9] Philip J. Sammons, Tomonari Furukawa, and Andrew Bulgin. Autonomous pesticide spraying robot for use in a greenhouse abstract. In *International symposium on intelligent robot system*, 1994.

[10] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Bradford Company, 2004.

[11] Sebastian Thrun, Michael Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia M. Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary R. Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara V. Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, 2006.