AN ABSTRACT OF THE THESIS OF

Jason Aaron Greco for the degree of Honors Baccalaureate of Science in Computer Science presented on August 19, 2010. Title: Automatically Generating Solutions for Sokoban Maps.

Abstract approved:

____

Alan Fern

Generating solutions to Sokoban levels is an NP-hard problem that is difficult for even modern day computers to solve due to its complexity. This project explores the creation of a Sokoban solver by eliminating as many potential moves as possible to greatly limit the overall search space. This reduction is achieved through abstracting out moves that don’t involve a box push and also through removing any deadlocked states where the problem is in a state that makes it unsolvable. The result is a Sokoban solver that shows substantial improvement over a standard exhaustive approach.

Key Words: Sokoban, Artificial Intelligence, Search

Corresponding e-mail address: grecoj@onid.oregonstate.edu

Automatically Generating Solutions for Sokoban Maps

by

Jason Aaron Greco

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science

Presented August 19, 2010
Commencement June 2011

Acknowledgments

I would like to thank my Honors Thesis Chair, Alan Fern, PhD, and my Honors Thesis Committee Members, Mike Bailey, PhD and Prasad Tadepalli, PhD from Computer Science, for their time, support, and feedback during the completion of this honors thesis.

I would also like to thank my family for their ongoing patience and support throughout this project.

Table of Contents

Page

Introduction....1

Thesis Statement....2
Background....2
Significance....3

Methodology....5

Exhaustive Search Method....5
Improved Method....5
The Box Pushes....5
Duplicate Checking....6
Deadlocks....8
Reconstructing the Winning Path....10
Summary....10

Results....11

Deadlock Removal....11
Comparing the Methods....13
Summary....18

Conclusions....19

Limitations....19
Future Work....19

Bibliography....21

List of Figures

Figure Page

1 First level of the original Sokoban game 2

2 Two states with different sets of movable locations...................................7

3 Box trapped in a corner..8

4 Two boxes trapped against a wall..9

5 Deadlocked states on Microban level 10 11

6 Microban levels 2 and 10 with deadlocks removed..................................13

List of Tables

Table Page

1 Number of box locations before and after deadlock removal....................12

2 Number of generated states before finding a solution 15

3 Execution times in ms for levels 1-20 with each of the solvers.................16

4 Speedup over standard breadth-first search method 17

Introduction

Non-deterministic polynomial hard, also referred to as NP-hard, problems pose a difficult challenge in solving them because current computing solutions require an exponential running time to successfully find a solution in worst case scenarios. Algorithms of exponential complexity are effectively useless because of the limitations of computational power, and it is necessary to rely on alternative methods to exhaustive search by utilizing clues from the input to reduce the overall search space (Dasgupta, Papadimitriou, & Vazirani, 2006).

Although these problems are very difficult for machines to solve, it is often possible for humans to do some of them in a much shorter time frame. The game Sokoban is a great example of an NP-hard problem and the task of generating solutions to its problems is of interest for study because it doesn't have a large number of rules yet still remains a complex problem (Dor & Zwick, 1999). Despite its simple rules, Sokoban has a very high branching factor, and many optimal solutions can require upwards of 600 moves to solve, making exhaustive search extremely difficult (Botea, Muller, & Schaeffer, 2003).

This project explores the possibility of creating a program capable of solving puzzle maps for Sokoban. The effects of abstracting movement to the individual boxes from the solution algorithm are analyzed for their impact on both the speed of finding a solution, and its level of optimization. Also covered are the effects of checking for different sets of trapped moves and how removing them from potential solutions affects the overall performance of the solver.

Problem Statement

The purpose of this research is to design and implement an algorithm capable of generating a solution to Sokoban maps.

Background

Sokoban is a single player puzzle game first created in 1982 which involves moving a set of boxes onto a set of goal platforms by strategically navigating them through a two-dimensional maze. The words "soko ban" are Japanese for "warehouseman" (Wagner, 1988) and the game "Sokoban is analogous to the problem of having a robot in a warehouse move specified goods from their current location to their final destination, subject to the topology of the warehouse and any obstacles in the way" (Junghanns & Schaeffer, 2001, p. 220).

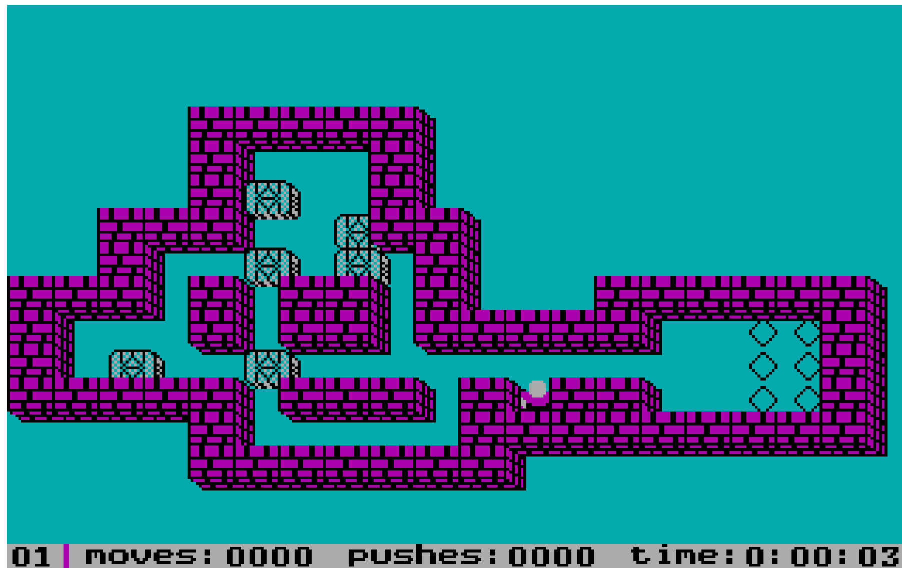

Figure 1: First level of the original Sokoban game

In Sokoban a map is composed of both *wall squares*, which cannot be occupied by either a player or a box, and *free squares*, which can be freely occupied by both boxes and the player. The objective of the game is to move all of the boxes onto the set of *goal squares*. A *move* is defined as when the player moves to a location that is adjacent to the

current position, while a *push* is when the player pushes one of the boxes over one location. The player is only allowed to push one box at a time and can only push the boxes, not pull them.

The problem of generating a solution to a Sokoban map has been proven to be PSPACE complete (Culberson, 1997) indicating that it is also in the set of NP-hard problems. Because of the complexity of such problems it becomes necessary to find more specialized solutions than what many of the more general algorithms can offer by themselves. One of the major challenges in solving Sokoban levels is the problem of trapped moves, also known as deadlocks. It is possible for the player to push a box into a location such that it then becomes impossible for the level to be solved. Despite being unable to solve the problem, unless the search algorithm is aware it has reached a dead end, it will continue to look for potential solutions by moving around the non-trapped boxes effectively wasting search time (Takes, 2007).

Significance

No polynomial time algorithm has yet been discovered for NP-hard problems such as generating Sokoban solutions, making even moderately difficult solutions too time consuming to solve through traditional means (Dasgupta, Papadimitriou, & Vazirani, 2006). This research aims to find ways of improving the efficiency to the NP-hard problem of generating solutions to Sokoban problems so that it becomes possible to solve them in a realistic timeframe. The problem of generating solutions to Sokoban maps is a good platform for developing the tools involved in solving these more advanced problems in Artificial Intelligence because of its simple design and high branching factor.

One of the key advantages humans have over computers is the ability to utilize past knowledge gained from previous similar puzzles rather than starting each new level from scratch. This project could potentially help lay the foundation for more advanced Sokoban solvers that utilize machine learning techniques to simulate a person's reasoning abilities.

Methodology

Breadth-First Search Method

A basic method of generating a solution to a Sokoban problem is through a breadth-first search algorithm. This method searches through every possible move the player could make until it finds a state in which all of the goal squares are occupied by boxes. While it could solve any problem on a theoretical computer with unlimited processing power and storage, this method is very impractical without any extra logic to reduce the number of potential moves. A standard breadth-first search method is used in this project as a test benchmark in order to objectively measure the differences in other methods against it.

Improved Method

The improved method used in this solver combines the effects of abstracting man-moves through a push() method, duplicate checking, and checking for deadlock states to eliminate. This combination greatly reduces the overall number of potential moves making it possible for the solver to solve Sokoban problems within a realistic timeframe. This method works out well because lots of Sokoban levels get their challenge by making many of the player's moves result in a deadlocked state leaving only a few moves that will actually help advance towards a solution.

The Box Pushes

One of the most costly parts of finding a solution in Sokoban is the extremely high number of ineffective moves the player can make while moving between boxes, where an ineffective move is one that doesn't either push a box or advance the player closer to another box. Because of the huge number of potential moves in Sokoban, these

ineffective moves can use up very large amounts of the computer's resources exploring all of these different paths.

Abstracting the movements between boxes from the automated solver greatly reduces the overall number of possible moves. The reason this is possible is, due to the nature of the game, there will never be a necessary move that is not either a box push or a move to another box.

In order to find optimal paths to boxes, the solver utilizes Dijkstra's algorithm to calculate shortest distances to everywhere on the map. The algorithm runs in O(E*log(V)) time, where V represents the number of open locations on the map and E is the number of potential moves between locations, and is only run once for any given state. After running Dijkstra's algorithm on the map it not only lets the solver know which different boxes it has access to, but also provides a shortest-distance path to that box. This, along with a quick check to see whether or not the spot the box is being moved to is open, allows the solver to know all possible moves that could be made from the current player location.

Duplicate Checking

One of the most important parts of the solver is duplicate checking. Without it the solver could potentially alternate between the same set of moves indefinitely and never actually find a solution. By introducing duplicate checking it also allows the solver to eliminate a very large number of redundant checks on moves that have already been checked. Although it is necessary, the cost can becomes very high as more and more different potential moves are explored. To minimize its impact on the solver, all of the different moves that have been explored are added to a hash table using Java's HashMap

in the java.util library.

A hash table was chosen as the data structure to store each visited state because it has both an efficient add() and an efficient contains(). A new item is added to the hash table every time a new non-duplicate move is discovered and is completed in constant time. Although many data structures have constant time adds, a hash table is ideal because it also has a contains method with an average time of O(1) which is important because it is used whenever a non-deadlocked move is discovered (Drake, 2005).

Another form of duplicate checking is to remove instances where the boxes are all in the same locations but the player's location is different. An example of this duplicate would be if a player pushed a box left and then immediately pushed it right. In this instance the player would be on the right of the box after the first push, and on the left of the box after the second. This can be achieved by ignoring the player's current location when storing a state for duplicate checking. In a few instances, such as the example in Figure 2, it is actually necessary to differentiate between these scenarios, which is done by comparing the sets of movable locations for each of the different states. If the sets of movable locations are equal then the state is a duplicate and it is then ignored, otherwise it is not a duplicate and the state is added to the set.

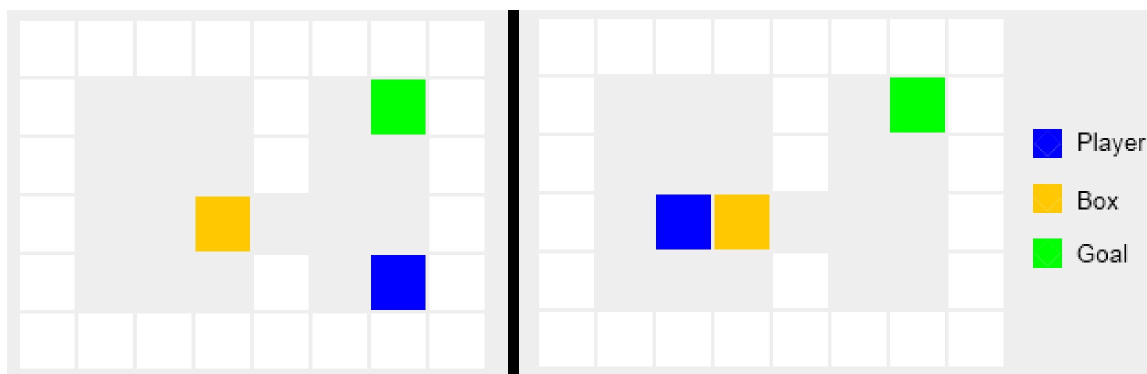

Figure 2: Two states with different sets of movable locations

Deadlocks

In Sokoban there are many states known as *deadlocks* where a box has been moved to a position that does not allow the player to successfully complete the level. These deadlocks are potentially very costly to the solver because, unless it is aware of its inability to proceed, it will continue attempting to move other boxes around effectively wasting large amounts of resources. The actual process of checking for deadlocks is theoretically just as difficult as the actual solving of the level because there is likely no efficient way to detect all possible forms of deadlock. In the detection of deadlocks it is important to distinguish between two different possible types: single block deadlocks and multiple block deadlocks.

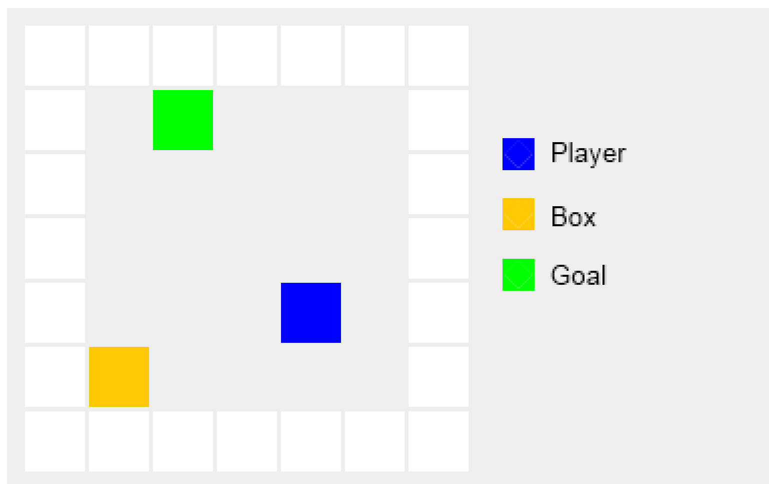

Figure 3: Box trapped in a corner

Single Box Deadlock:

A single box deadlock is a position on the map where if any box is placed it is unable to be moved onto a goal platform. Figure 3 shows an example of one such deadlock where a box is trapped in the corner and can no longer be pushed by the player. Another example of this type of deadlock would be when a box is up against a wall without a goal square on that wall because there is no way to push the box off of the wall.

Because these deadlocks only involve a single box, the solver can check for these potential hazards before actually beginning to solve the problem. Doing this at the beginning significantly reduces the overall cost of deadlock checking because the major checks only need to be done once. The solver checks each square on the map to see if a box will be trapped when placed there, and then during the solution algorithm only needs to perform a single check to see if the box can be moved to that location. This method reduces the cost of this form of deadlock detection to a time of O(1) after the initial one time check.

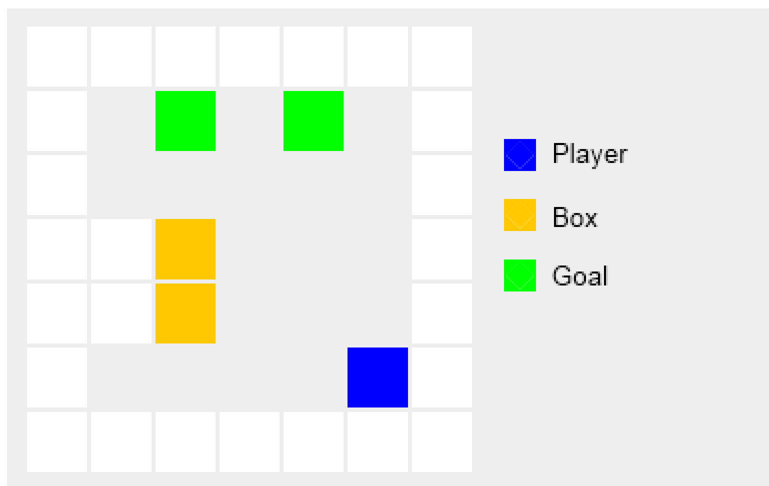

Figure 4: Two boxes trapped against a wall

Multiple box deadlock:

These forms of deadlocks differ from the single box deadlocks because they also depend on the locations of other boxes in the level. Figure 4 illustrates an example of this form of deadlock where the two boxes up against a wall create a deadlock because neither of the boxes can now be moved by the player. These deadlocks are more costly to detect than those involving only one box because they change each time a box is moved so they cannot be simply checked at the beginning and must instead run through a series of checks after every time a box is pushed. To reduce the cost of this form of deadlock

detection only deadlocks involving the last pushed box are checked for. Because the rest of the map remains unchanged, it can be assumed that no new trapped boxes will be introduced that do not include the most recently moved box. By only checking around the specific box, the overall time required to check for these deadlocks is reduced to a constant time operation.

Reconstructing the Winning Path

Without the ability to store the path leading to a solution the Sokoban solver would not be very useful. In order to accomplish this with minimal impact to the overall running time of the algorithm, each major state occurring immediately after a push stores both the previous major state and the path the player had to take to move between the two. After a solution is found the solver traces through all the previous states until it reaches the beginning and then reverses all of those to construct the path taken.

Summary

This project combines all of the effects of utilizing box pushes, removing duplicates, and checking for deadlocks to create a usable Sokoban solver. By reducing the total number of potential moves through these methods, it decreases the large branching factor allowing the solver to effectively find a solution.

Results

To test the overall performance of the Sokoban solver both the effectiveness of removing deadlocked spaces and the overall performance are analyzed. A package of small Sokoban problems known as Microban was used to test the effectiveness of different versions of the solver. Despite their smaller size, the Microban levels were all designed to demonstrate at least one or two different concepts in Sokoban (Skinner, 2000).

Deadlock Removal

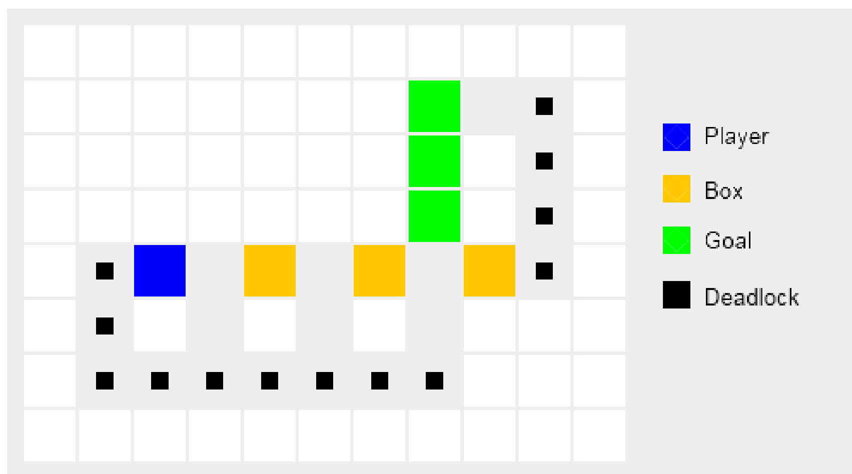

Figure 5: Deadlocked states on Microban level 10

The removal of trapped states from potential solutions is very important to the Sokoban solver. This greatly reduces the total number of movable spaces making the solver capable of turning a level into a much more manageable size. This process can be seen working in Figure 5,which depicts Microban level 10, with each deadlock that has been detected by the solver marked with a black square to signify that a box cannot safely be pushed to that location. Table 1, shown below, lists the total number of potential box locations before and after deadlock removal for each of the first 20 problems in the Microban pack.

Level	Total Box Locations	Non-Deadlock Box Locations	Percent Reduction
1	14	9	36%
2	19	5	74%
3	20	11	45%
4	20	8	60%
5	27	12	56%
6	28	14	50%
7	30	12	60%
8	27	15	44%
9	13	8	38%
10	27	14	48%
11	25	13	48%
12	23	9	61%
13	20	14	30%
14	16	6	63%
15	19	8	58%
16	31	21	32%
17	17	9	47%
18	22	13	41%
19	19	11	42%
20	23	12	48%
Average	**22.0**	**11.2**	**49%**

Table 1: Number of box locations before and after deadlock removal

With an average reduction in potential box locations of almost 50%, the deadlock removal is effective in reducing the total number of moves the solver must check to find a solution. The worst case level, number 13, still has its potential locations reduced by almost a third of their total, and in level 2 the deadlock detection is able to remove almost three quarters of the total box locations. Figure 6 shows a comparison of these two levels with deadlocked states marked in black. Level 2 is able to remove so many spaces because there are no goal squares along any of its major walls, while level 13 is only able to remove a few of its corner squares as deadlocks.

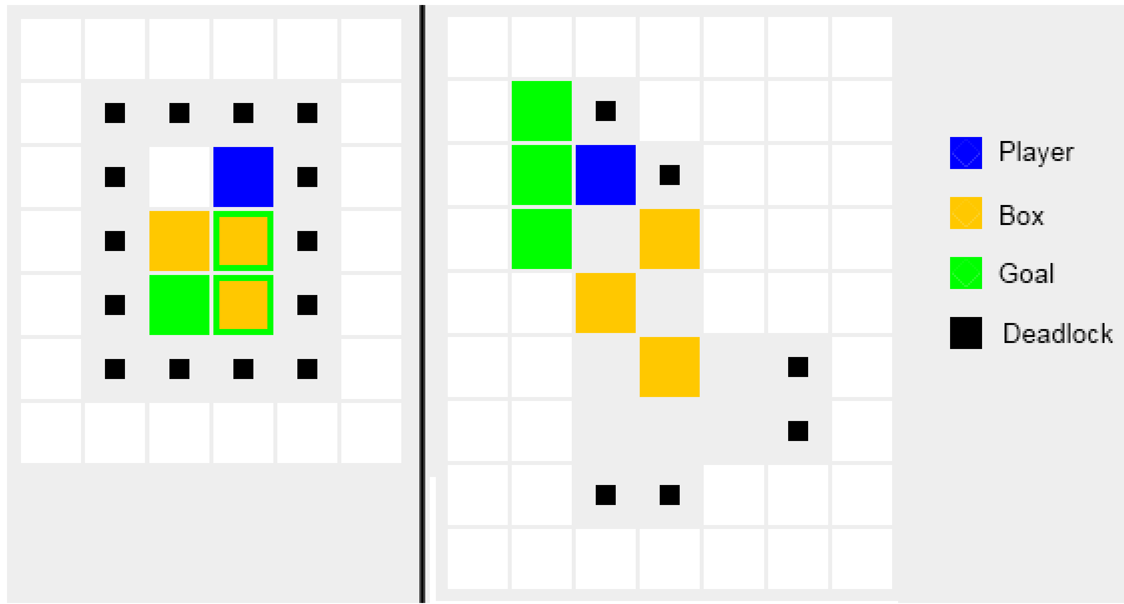

Figure 6: Microban levels 2 (left) and 10 (right) with deadlocks removed

Comparing the Methods

To gain an accurate measurement for how effective a given Sokoban solver is the measurement should reflect the time required to execute. The only portion of the Sokoban solver that doesn't require a constant time is the check to see if a particular state has already been explored. Although this is made as efficient as possible through the use of a hash table, it is still the slowest part of a Sokoban solver. Because of this, the efficiency of an algorithm can be determined by both the number of states that have been generated searching for a solution and the overall running time required to generate a solution. The following algorithms were tested to see how many states were generated before finding a solution to each level:

- *Standard breadth-first search* – Performs an exhaustive search of every possible move using a breadth-first search with duplicate detection until it finds a solution. This is the most basic of the algorithms and involves minimal logic.

- *Breadth-first search with deadlock prevention* – Same as the standard breadth-first search, however also adds checks to prevent boxes from being moved into known deadlock scenarios.
- *Push method* – Utilizes box pushes rather than individual man moves along with duplicate detection and does not have any form of deadlock prevention.
- *Push method with deadlock prevention* – Combines the box push method with deadlock prevention and will always be the best of the four algorithms.

Table 2 shows a comparison of each of the different algorithms tested to see how many different states were generated before finding a solution. Levels 5 and 7 were not included in the calculation of any of the averages because they took more than 500,000 generated states to solve with the standard breadth-first algorithm that did not utilize deadlock detection which would have required excessive storage space and running time to let run to completion.

Level	Standard Breadth-First Search	Breadth-First With Deadlock Detection	Push Method No Deadlock Detection	Push Method With Deadlock Detection
1	1,404	811	101	34
2	2,076	298	17	7
3	4,178	2,117	252	116
4	49,120	4,429	4,062	161
5	>500,000	78,256	20,203	1,420
6	139,803	25,434	8,810	1,058
7	>500,000	161,501	101,218	4,255
8	9,394	4,476	397	180
9	1,515	581	95	16
10	13,628	5,012	781	276
11	13,354	5,383	608	186
12	5,945	1,024	153	27
13	45,124	17,654	3,447	825
14	2,400	590	169	24
15	2,851	935	138	41
16	240,957	96,032	13,602	3,753
17	11,863	1,749	1,159	89
18	10,564	3,888	456	160
19	3,848	1,900	244	120
20	8,950	2,700	465	139
Average	**31,499**	**9,723**	**1,942**	**401**

Table 2: Number of generated states before finding a solution (averages do not include levels 5 and 7)

Deadlock detection by itself reduced the total number of generated states by 69.1%, while utilizing box pushes reduced the number of states by 93.8%. Combining these effects resulted in a reduction of generated states of 98.7% from the original solver. Level 2 was by far the fastest, requiring only 7 generated states to find a solution with the final solver, which can most likely be credited to it having the highest number of deadlocked states removed, with 74% of its box locations having been removed.

Because of the increased cost per state incurred from both deadlock detection and the shortest path computations, it is also important to look at the actual running time of each algorithm to make sure that these don't offset the benefits of the reduction in states

generated. Table 3 shows the total running time (in milliseconds) required to find a solution for each of the solvers.

Level	Standard Breadth-First Search	Breadth-First With Deadlock Detection	Push Method No Deadlock Detection	Push Method With Deadlock Detection
1	56	39	13	8
2	60	10	1	1
3	120	61	17	10
4	1,501	126	122	12
5	N/A	3,470	732	58
6	4,938	1,152	367	52
7	N/A	7,556	3,811	177
8	488	304	18	12
9	69	11	2	1
10	508	182	28	13
11	450	178	19	7
12	198	28	5	1
13	1,381	564	125	31
14	81	11	4	1
15	66	21	4	2
16	11,080	4,430	548	194
17	397	34	25	5
18	298	100	13	5
19	100	176	8	3
20	260	72	55	7
Average	**1225.1**	**416.6**	**76.3**	**20.3**

Table 3: Execution times in ms for levels 1-20 with each of the solvers

As can be seen in Table 3, the decrease in running time of the improved solver is slightly less dramatic than the decrease in number of states generated, however there is still a significant improvement. Table 4 illustrates how much more effective each of the methods is over the standard breadth-first approach by showing their speedup on each level. Each value represents how many times faster the given solver was than the standard solver when comparing their execution times. These values could not be

computed for levels 5 and 7 because there was insufficient data due to their excessive run times on the standard breadth-first solver.

Level	Breadth-First With Deadlock Detection	Push Method No Deadlock Detection	Push Method With Deadlock Detection
1	1.4	4.3	7.0
2	6.0	60.0	60.0
3	2.0	7.1	12.0
4	11.9	12.3	125.1
5	N/A	N/A	N/A
6	4.3	13.5	95.0
7	N/A	N/A	N/A
8	1.6	27.1	40.7
9	6.3	34.5	69.0
10	2.8	18.1	39.1
11	2.5	23.7	64.3
12	7.1	39.6	198.0
13	2.4	11.0	44.5
14	7.4	20.3	81.0
15	3.1	16.5	33.0
16	2.5	20.2	57.1
17	11.7	15.9	79.4
18	3.0	22.9	59.6
19	0.6	12.5	33.3
20	3.6	4.7	37.1
Average	**4.5**	**20.2**	**63.1**

Table 4: Speedup over standard breadth-first search method

On average the deadlock prevention appears to speed up the solver around 3 to 4 times by turning it on, and abstracting out the individual man-moves sped up the solver by a factor of 20. The final algorithm can find a solution to a Sokoban level on average 63 times faster than the exhaustive approach, indicating a rather large improvement in execution time for the solver.

Summary

The Sokoban solver developed in this project shows definite improvement over the exhaustive approach. With 49% of potential box locations eliminated as deadlocks and an average speedup of 63 the new solver shows significant speedups in smaller Sokoban problems. Because of Sokoban's complexity solving problems that are much larger and more difficult would likely require combining the techniques of this solver with more advanced methods.

Conclusions

Generating solutions to Sokoban problems presents an interesting challenge because it is possible that no truly optimal solution will ever be discovered. The large number of potential moves in Sokoban problems makes exhaustive search impractical, and solvers must instead rely on other methods to go about this challenge. This project has explored the ideas of both looking at only box pushes rather than man moves and the removing of deadlocked states as a means of reducing the overall number of potential moves the solver must explore in finding a solution. The solver created shows substantial improvement over the original breadth-first search, although it could still be adapted to utilize more advanced heuristics to help further improve its efficiency.

Limitations

The difficulties encountered by this Sokoban solver appear to be with problems that are either very large or have lots of boxes. The challenge with large problems comes from the large branching factor and with lots of open space that can't be eliminated through deadlock prevention. Consequently, it becomes difficult for the solver to process all of the different potential moves. Levels with large numbers of boxes are also difficult because it reduces the effectiveness of abstracting out the man-moves.

Future Work

The most time consuming portion of the Sokoban solver is the process of checking to see if a particular state has already been explored. Despite using the relatively efficient hash table this operation is still the only part of the solver that becomes slower the longer the program is run. If this process could be sped up in any

way it would provide a significant boost to the solver's ability to function on larger problems.

Another possibility for future research with Sokoban solvers would be to combine them with a form of machine learning to help the solvers recognize patterns and learn what does and does not work well. This form of solver could also read through replays of humans solving Sokoban problems and analyze their methods in an attempt to recreate similar problem solving strategies.

The solver could also be adapted to find an optimal solution requiring the fewest possible number of man moves. This presents an interesting challenge because the algorithm would be unable to look at only the box pushes, forcing the solver to find different ways of reducing the total number of available moves.

Bibliography

Botea, A., Muller, M., & Schaeffer, J. (2003). Using Abstraction for Planning in Sokoban. *Computers and Games*, 360-375.

Culberson, J. C. (1997). *Sokoban is PSPACE Complete.* Department of Computing Science, University of Alberta.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2006). *Algorithms.* McGraw-Hil.

Dor, D., & Zwick, U. (1999). Sokoban and Other Motion Planning Problems. *Computational Geometry*, 215-228.

Drake, P. (2005). *Data Structures and Algorithms in Java.* Upper Saddle River: Prentice Hall.

Junghanns, A., & Schaeffer, J. (2001). Sokoban: Enhancing general single-agent search. *Artificial Intelligence*, 219–251.

Skinner, D. W. (2000, April). *Microban*. Retrieved August 1, 2010, from http://users.bentonrea.com/~sasquatch/sokoban/index.html

Takes, F. (2007). *Sokoban: Reversed Solving*. Leiden University.

Wagner, R. (1988). Puzzling Encounters. *Computer Gaming World*, 42–43.