De La Salle University - Manila
College of Computer Studies

MCO1: Sokobot

In Partial Fulfillment of the Course Requirements in CSINTSY - S17
Term 1, AY 2023 - 2024

Submitted by:
GROUP : Intzy
Gutierrez, Allen Andrei D.
Palacios, Jeremy Koen K.
Sabater, Cassandra Jo T.
Thay, Jerrel Chadler O.

Submitted to:
MR. TIAM-LEE, THOMAS JAMES

October 23, 2023

I. INTRODUCTION

In a world where intelligent algorithms are constantly pushing the boundaries of what is possible, the realm of puzzle solving has always been a captivating and intellectually stimulating arena. The Sokoban puzzle, with its maze-like structures and crate-shattering puzzles, serves as a benchmark of algorithmic prowess.

To this end, the researchers embarked on a journey to develop an algorithm capable of tackling all the challenges Sokoban has to offer. The report will focus on examining the design of the algorithm that was created. In addition, the report will present the implementation and performance evaluation of the algorithm. The primary approach to solving these puzzles involves using search algorithms. These algorithms systematically explore, evaluate, and prioritize various moves, resulting in an optimized quest for a solution.

II. ALGORITHMS

The algorithm used to solve the sokoban puzzles is Greedy Best-First Search (GBFS), which is a heuristic search algorithm that finds the most promising path from a given starting point to a goal. This algorithm is used as it satisfies the objective of finding a solution to solve the puzzle with the least possible moves the bot can think of, given the time limit of 15 seconds for the bot to think.

STATE REPRESENTATION

In the provided code, a state in the Sokoban puzzle is represented as an instance of the State class. This representation captures essential information about the puzzle's current configuration. The player's position is denoted by the "coordinateX" and "coordinateY" instance variables, specifying the x and y coordinates of the player on the grid. The positions of the boxes are stored in the "boxesX" and "boxesY" ArrayLists, where each element in these lists corresponds to the x and y coordinates of a box, respectively. Additionally, the "heuristic" instance variable quantifies the heuristic value associated with the state, serving as an estimate of how close the state is to a goal state, crucial for guiding search algorithms. The "path" StringBuilder instance variable keeps track of the path leading to the current state, representing a sequence of actions or moves needed to reach this state from the initial state. This representation allows for efficient manipulation and evaluation of Sokoban puzzle states, enabling search algorithms to navigate the puzzle space effectively and find a solution by making informed decisions based on the stored information.

Furthermore, the State class offers various methods to access and manipulate these state attributes. For example, it provides getter and setter methods for the heuristic value and the depth of the state within the search tree. Customized methods, such as "copy()", allow for the creation of duplicate states, enabling exploration of different possibilities while retaining the essential characteristics of the original state. To add to this, it includes methods to

determine if a box exists at a given coordinate and retrieve its index, calculate A* search costs by incorporating the heuristic value, sort the box positions lexicographically, and compare the two states for equality.

REPRESENTING THE SEARCH PROBLEM

The SokoBot class stores the algorithms and methods required to solve the Sokoban problems. It encapsulates the essential functionalities for solving Sokoban puzzles and utilizes the State class to represent the current state of the puzzle. The initialization process involves identifying the puzzle's starting state and goal states. Specifically, the "createStart()" method determines the initial positions of the player and boxes, while the "createGoal()" method sets the goal state by identifying the target position for the boxes.

- PRUNING DEADLOCK STATES

In order to avoid unnecessarily searching "hopeless" states, the code also includes deadlock detection functionality that identifies areas on the puzzle map where boxes can become trapped and unmovable, making it impossible for the algorithm to reach the goal state. The detection of deadlocks is primarily done by the method named "findDeadlocks()". The method iterates through the map, examining each position for conditions that indicate a deadlock. It checks for four corner conditions which include the upper-left, upper-right, lower-left, and lower-right corners of potential deadlocks. If the conditions for a deadlock are met at a specific position, its coordinates are added to a HashSet called "deadSquares". By analyzing the entire map, the "findDeadlocks()" method creates a comprehensive list of coordinates that represent the corners of potential dead zones. These coordinates are then utilized to determine if a given position is a deadlock by searching for it within the deadSquares set. If a position is found within this set, it is considered a deadlock , and the program takes the necessary precautions to avoid these positions during the puzzle-solving process.

- SEARCH ALGORITHM

The core of this code is the Greedy Best-First Search (GBFS) algorithm, which is implemented in the "gbfs()" method. GBFS maintains a priority queue of states, ordered by their heuristic values, and explores potential states. When a goal state is found, the search terminates, and the solution path is returned. In order to explore these states the "exploreState()" method was implemented. This method systematically explores potential next states of the puzzle, given the current state represented by State s and a set of explored states. To achieve this, the method first identifies the player's current position using coordinateX and coordinateY. It then evaluates the feasibility of four potential moves - up, down, left, and right - by examining the map's layout. The method ensures that the Sokoban puzzle rules are strictly enforced, allowing boxes to be moved only when conditions permit and avoiding potential deadlock conditions. New states representing these moves are

generated and added to the toExplore list. The method also checks whether any of these newly generated states correspond to the goal state, where all boxes are correctly positioned. Ultimately, the toExplore list is returned, containing these generated states as candidates for further exploration. The method references the explored set to avoid revisiting states already explored, contributing to the efficiency and effectiveness of the search for an optimal Sokoban puzzle solution.

The decision to incorporate Greedy Best First-Search (GBFS) with the Manhattan distance heuristic in the provided Sokoban puzzle-solving code is based on the efficiency and effectiveness of this combination in solving combinatorial search problems. GBFS was chosen for its ability to prioritize states based on heuristic values, which in this case, are determined by the Manhattan distance.

The computeHeuristic() method calculates the Manhattan distance heuristic, which estimates the proximity of a given state to the goal by measuring the distance between each box and its corresponding target position. This heuristic is particularly suitable for the Sokoban puzzle, as it aligns with the intuitive understanding that a shorter distance for boxes to reach their goals indicates progress towards a solution.

The synergy between GBFS and the Manhattan distance heuristic enhances the search process in multiple ways. Firstly, GBFS selectively explores states that are believed to be closer to the goal, which is crucial in the Sokoban puzzle due to its extensive search space. By prioritizing states with lower heuristic values, the algorithm is directed towards more promising solutions. This selective exploration significantly reduces the number of states that need to be considered. resulting in a more efficient search.

Furthermore, the combination of the Manhattan distance heuristic and the GBFS aligns with the principle of informed search, where the algorithm utilizes domain-specific knowledge to make intelligent choices, thereby expediting the puzzle-solving process. Although GBFS does not guarantee an optimal solution, it is proficient at finding satisfactory solutions quickly.

III. EVALUATION AND PERFORMANCE

For every level solved by the Sokobot, various statistics were monitored including the total number of states generated by the search tree, redundant states, explored nodes and the remaining number of nodes in the frontier upon arriving at a solution. These statistics provide insight into the efficiency and performance of the solver. The Sokoban solver was tested across various layouts, solving the majority of the given levels within the 15 second time limit.

	No. of Generated States	No. of Redundant States	No. of Nodes in Frontier upon Termination	No. of Explored Nodes upon Termination	No. of moves the solution returns	Time taken on average of >=5 runs (in seconds)
testlevel (4 boxes)	507	36	17	394	37	0.070s
twoboxes1	136	3	22	97	29	0.056s
twoboxes2	434	9	21	434	48	0.053s
twoboxes3	415	20	20	307	62	0.060s
threeboxes1	2789	126	97	1975	98	0.122s
threeboxes2	7433	256	128	5553	153	0.194s
threeboxes3	791	22	33	615	95	0.082s
fourboxes1	6546	565	47	4947	95	0.193s
fourboxes2	32760	2030	409	23809	200	0.426s
fourboxes3	67468	3180	289	49783	235	0.560s
fiveboxes1	27101	1306	494	20403	124	0.450s
fiveboxes2	24720	1821	221	18925	184	0.405s
fiveboxes3	285211	21681	1418	203771	294	1.742s
original1	370695	11229	753	295457	311	2.480s

*Maps unsolved within 15 seconds: original2, original3

As shown in the table, there is a proportional relationship between the number of boxes and the number of states generated. “Fourboxes3” and “fourboxes2” generated a relatively larger number of states due to their wider surface areas — fourboxes3 has a height and width of 9 & 10 respectively, while fourboxes2’s dimensions are 7 & 10. The same can be said for “threeboxes1” and “threeboxes2”. The bigger the map and the more boxes there are, the more states the bot has to create and traverse to, needing more time to return a solution.

Compared to its counterpart i.e. A* search, using GBFS paired with a Manhattan distance heuristic returns a solution faster as it involves less computation and memory usage. This is due to its greedy nature of selecting the node with the lowest heuristic. However, it doesn’t return the optimal solution with the least number of moves. For instance, it is possible

to solve the “fourboxes2” level in 160 moves, but the Sokobot returns a solution with 200 moves as seen below.

“fourboxes2” solved within 160 moves

“fourboxes2” solved by the Sokobot

IV. CHALLENGES

One of the challenges faced in the project was creating functions or lines of code that checks possible moves, generates the state of those moves that can be added to the frontier and explores those states. This was a challenge as every situation had to be considered which means lots of checkers of different scenarios.

Another challenging part of the project is to find a way to reduce the time needed for the algorithm to come up with a solution. Possible reasons why the algorithm takes a significant amount of time or isn't able to solve a puzzle are the algorithm searches through redundant states, it explores states that end up in a deadlock situation or the branching factor of some levels or stages are too big for the algorithm to solve in a certain amount of time. This is evident in levels that are big (height and width of the map) and have more than 4 boxes as the number of generated states increase as these factors increase. This is why the group implemented GBFS, the Manhattan distance heuristic and a deadlock detector which checks if the boxes are in a corner of the current state to help the bot explore the states optimally. Among the implementations, the deadlock detection was the hardest to implement as the group had to figure out the possible deadlocks and find a way to represent it through code.

V. CONCLUSION/REFLECTION

This project gave the group an opportunity to play sokoban manually, analyze the states of the game and discuss what algorithm would be most effective to find a solution for sokoban puzzles. The group decided to implement greedy best first search alongside the Manhattan distance heuristic and added functions and lines of code that checks and updates states. It was interesting to see how the search algorithm, the greedy search algorithm helps

find a solution in the shortest amount of time, the Manhattan distance heuristic supplying numerical information for the algorithm to make those decisions and how all of the other checkers and functions work all together to solve the puzzle. It is also interesting how the program is able to solve the puzzle that the members of the group couldn't solve manually. We realized how powerful these algorithms are in finding solutions not only in sokoban puzzles but possibly in other situations and problems. It is far from perfect in solving various problems but still impressively effective. Overall this was a fun and interesting project and it definitely showed the group the potential of intelligent algorithms.

CONTRIBUTIONS

Name of Member	Tasks
1. Gutierrez, Allen Andrei D.	Algorithms, Evaluation and Performance
2. Palacios, Jeremy Koen K.	Code, Introduction, Algorithms
3. Sabater, Cassandra Jo T.	Code, Evaluation and Performance
4. Thay, Jerrel Chadler O.	Evaluation and Performance, Challenges and Conclusion

REFERENCES

AI | Search Algorithms | Greedy Best-First Search | Codecademy. (2023). Codecademy. https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search

Educative. (n.d.). *Educative Answers - trusted answers to developer questions*. https://www.educative.io/answers/what-is-manhattan-distance-in-machine-learning

GeeksforGeeks. (2023). *Greedy Best first search algorithm*. https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

Great Learning Team. (2022). *Best First Search Algorithm in AI | Concept, Algorithm and Implementation*. https://www.mygreatlearning.com/blog/best-first-search-bfs/

Russell, S. & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach, 4th Ed. New Jersey: Pearson*

Sokoban Wiki. (n.d.). *Retrieved from http://sokobano.de/wiki/index.php?title=Solver*