CSINTSY S12 AY 2023-2024

Members: Campos, Chan, Clemente, Isolan, Lingat

I. **Brief Introduction:**

The project aims to solve as many Sokoban problems as possible within the time limit of 15 seconds. Sokoban is a Japanese warehouse game where the goal is to put all the crates in the marked spots. Although the rules can be very simple, trying to achieve the goal is hard because of some conditions like not being able to pull a box or push two boxes at the same time. The project uses state-based algorithms to search for the solution of a given Sokoban puzzle.

II. **SokoBot Algorithm**:

1) State Representation

The Sokobot Algorithm is designed to tackle the complex Sokoban puzzle by employing a state-based approach. The algorithm's core foundation lies in the representation of these states. Each state encapsulates information, including the dimension of the puzzle, coordinates of the player, position of the crates, location of the goals, map layout, and a heuristic value. This state representation provides a dynamic foundation for the algorithm to understand the puzzle's current configuration.

In creating the initial state, the algorithm uses a constructor that initializes these key attributes. The puzzle map input is examined to establish the player's position and location of crates, while the goals are determined using the map's layout. A heuristic value is then calculated to assess the distance between crates and their respective goals. This initial state serves as the starting point for the algorithm's process throughout the puzzle.

```
public State(int height, int width, char[][] mapData, char[][] itemsData) {
        this.height = height;
        this.width = width;
        this.player = findPlayer(itemsData);
        this.boxes = findBox(itemsData);
        this.goals = findGoal(mapData);
        this.mapData = mapData;
        this.itemsData = itemsData;
        this.heuristic = calculateHeuristic();
        this.parent = null;
        this.moveSequence = "";
    }
```

As the algorithm progresses, another constructor is used in the method to update the state in response to the player movements. The parent state and a specific move (left, down, up, right) are inputs for this constructor. It then computes the new state by taking the player's move on crate positions and their connection to the goals into account. The updated state preserves the puzzle's dimension and layout, recalculates the heuristic value, aiding on the exploration of state space and the reconstruction of the solution path.

```
public State(State parent, char move) {
        this.height = parent.getHeight();
        this.width = parent.getWidth();
        this.parent = parent;
        this.moveSequence = parent.getPath() + move;
        this.mapData = parent.getMap();

        char[][] updateData = computeNextState(parent, move);
        this.itemsData = new char[height][width];

        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                this.itemsData[i][j] = updateData[i][j];
            }
        }
        this.player = findPlayer(this.itemsData);
        this.boxes = findBox(this.itemsData);
        this.goals = parent.getGoal();
        this.heuristic = calculateHeuristic();
    }
```

2) Heuristic

An implementation of heuristics is crucial in the algorithm in order to guide the bot's decisions. The Manhattan Distance heuristic approach is used in the Sokoban puzzle to calculate the cost of achieving the desired state. This cost estimation involves two primary components.

It begins by calculating the minimum Manhattan distance for each crate to its closest goal. As the algorithm iterates through the crates and goals, it calculates the shortest distance between each crate and its corresponding goal. Following a comparison, the minimum distance is stored for this value. The sum of these minimal distances for each container serves as an estimation of the crate's distance from each of its individual destination spots.

```
private int manhattanDistance(XYBox box, Goal goal) {
        return   Math.abs(box.getX()   -   goal.getRow())   +   Math.abs(box.getY()   -
goal.getColumn());
    }
```

Consequently, the heuristic also takes into consideration the number of open goals in the puzzle. Open goals are those where crates have not yet been pushed to occupy a goal. The heuristic is enhanced to include the overall number of such open goals. This component displays the number of boxes that are still to be accurately placed at their corresponding goal locations.

```
private int calculateHeuristic() {
        int cost = 0;
        int min;
        int openGoals = goals.size();

         for (XYBox box : boxes) {
            min = Integer.MAX_VALUE;
            for (Goal goal : goals) {
                int dist = manhattanDistance(box, goal);
                min = Math.min(min, dist);
            }
            cost += min;
        }
        cost += openGoals;
        return cost;
    }
```

This approach aims to give the algorithm a reliable estimate of the amount of work still needed to complete the Sokoban puzzle. The algorithm efficiently progresses through the challenging tasks in an effort to find the best solution by prioritizing movements that reduce the estimated cost.

3) Search Algorithm (GBFS)

In order to find a solution, the SokoBot utilizes a Greedy Best-First Search Algorithm (GBFS). Similar to A* search, this algorithm relies heavily on the heuristic value to estimate the cost from the current state to the goal. Hence, it searches for the state with the lowest heuristic value which is considered closer to the goal state. With this in mind, the group implemented two data structures: a) openSet and b) visited which uses priority queue and hash set, respectively. The *openSet* stores states that are yet to be explored while the *visited* stores states that are already explored.

```
public SokoBot() {
        openSet = new PriorityQueue<>(new Compare());
        visited = new HashSet<>();
    }
```

In addition, a custom *Comparator* was also implemented to order the states based on their heuristic values. This simply compares the heuristic values wherein returning a positive or negative result will determine which state is prioritized.

```
public class Compare implements Comparator<State>{
    public int compare(State n1, State n2) {
        return n1.getHeuristic() - n2.getHeuristic();
    }
}
```

Before the search begins, it should start with an empty queue; thus, it clears first the open set and visited set then the initial state is added to the open set. While there are still states to be explored, the search will continue exploring states until it reaches the goal state. The explored states are all placed in the visited set to avoid duplicate visits. While the search continues, it simultaneously searches for the state with the lowest heuristic cost. Furthermore, the expandState was implemented to explore the newly generated states.

```
private String search(State initial, char[] moves) {
    openSet.clear();
    visited.clear();
    openSet.add(initialNode);

    while (!openSet.isEmpty()) {
        State node = openSet.poll();

        if (node.isGoalState()) {
            System.out.println(node.getPath());
            return node.getPath();
        }

        visited.add(node);
        expandState(node, moves);
    }
    return initial.getPath();
}

private void expandState(State node, char[] moves) {
    for (char move : moves) {
        if (node.isValid(move)) {
            State next = new State(node, move);
                    if    (!next.isDeadlock()    &&    !openSet.contains(next)    &&
            !visited.contains(next)) {
                openSet.add(next);
            }
        }
    }
}
```

4) Deadlock

In order to speed up the exploration of states, deadlock detection was utilized to determine if the move is valid or invalid. A deadlock situation can cause the player bot to stop moving because there are no possible

moves that the state can generate. The group implemented the most common deadlock scenarios such as the vertical deadlock and horizontal deadlock. The vertical deadlock happens when the player is surrounded by walls above or below the bot. On the other hand, the horizontal deadlock happens when the player is surrounded by walls from left and right.

```
 public boolean isDeadlock() {
    for (XYBox box : boxes) {
        int boxX = box.getX();
        int boxY = box.getY();

        if (isVerticalDeadlock(boxX, boxY) || isHorizontalDeadlock(boxX, boxY)) {
            return true;
        }
    }
    return false;
}
private boolean isVerticalDeadlock(int boxX, int boxY) {
    if (itemsData[boxX - 1][boxY] == '#' && itemsData[boxX + 1][boxY] == '#')
    {
        return true;
    }
    return false;
}

private boolean isHorizontalDeadlock(int boxX, int boxY) {
    if (itemsData[boxX][boxY - 1] == '#' && itemsData[boxX][boxY + 1] == '#')
    {
        return true;
    }
    return false;
}
```

5) Solution

Once all the boxes are in their target positions, this is counted as the goal state. The GBFS will backtrack the steps from the goal state to the initial state by using the parent-child relationship. This process reconstructs the solution path of the bot. The algorithm will return the solution path which represents the sequence of moves (left, right, up, down) generated from the search() to solve the Sokoban puzzle.

III. **Evaluation and Performance:**

The performance of the Sokobot Algorithm was evaluated through extensive testing given a variety of Sokoban puzzles of different sizes and complexities. This evaluation aims to reveal the strengths and weaknesses of the algorithm across several configurations.

In terms of puzzles with simpler layouts and fewer crates, it demonstrated its ability to find a somewhat optimal solution efficiently. For instance, solving the puzzle within the indicated time limit but on some maps uses more moves than the shortest path cost. However, as the complexity of the puzzle increased, it showcased an extensive time usage and was not able to produce a solution within the given time limit. An example map for this scenario is fourboxes2 and fourboxes3 which without a time limit, can make a solution in 31s and 40s respectively but is way out of the specified time limit for the solution generation.

Some strengths of the bot are smaller levels or medium sized levels with more space for the character to move about as the algorithm, when generating the graphs would generate less dead-locked positions and multiple moves can achieve the same goal. On the other hand, the weaknesses of the bot are larger levels as there will be more node generation thus will slow down the solution generation to the point that it will not reach a solution within the given time limit. It also does not perform well in situations where there is very little space to move in that nearly every node from the 2nd to n layer would be full of dead-locked nodes that take up more time and space, thus slowing down the solution output as well.

Below is a table containing the information of the maps and the time that the bot took to solve the problem:

Map	Time to Solve (in Seconds)
testlevel	0.04
twoboxes1	0.04
twoboxes2	0.04
twoboxes3	0.06
threeboxes1	0.10
threeboxes2	1.47
threeboxes3	0.10
fourboxes1	0.08
fourboxes2	TIME OUT
fourboxes3	TIME OUT
fiveboxes1	TIME OUT
fiveboxes2	10.82
fiveboxes3	TIME OUT
original1	TIME OUT
original2	TIME OUT
original3	TIME OUT

IV. **Challenges:**

A. How to represent each state in Java and graph it.

The first challenge we encountered was how to represent a single state in Sokoban. This is because there are many possible states in Sokoban and it is generally inefficient to put them all in one graph. Our first solution or algorithm to this problem was to create all the possible legal states from a given state and these new states will be the neighbor of the given state. The algorithm will continue until we have found the goal state. After implementing this algorithm, it has been found that it is very intensive and inefficient because there are so many possible states to the point that doing this approach is similar to just brute forcing the problem. We have also attempted to use an external library for graphing but we have found out that it may be inefficient to graph first and search later due to the large number of possible states. Upon realization, we have found out that it is best to generate and search the states along the way. How this works is that when evaluating the state, the algorithm checks first if the state is a goal state. If it is a goal state, it will stop searching and return the solution. If not, it will generate all the legal states derived from the state being evaluated and then they will be marked as a neighbor of their parent state and be added to the frontier of states that can be explored.

B. What possible algorithms could be used.

One of the challenges that came along with the project was coming up with what algorithms the group were going to use. Initially, the group brainstormed on what algorithms were possible to solve the problem. Among such algorithms were Breadth-First Search, Depth-First Search, Iterative Deepening Search, Greedy Best-First Search, and A* Search algorithm to name a few. We tried to first use uninformed searches because back then, we could not find a suitable heuristic for Sokoban. After finding out how to represent the states, we tried to implement Iterative Deepening Search (IDS) because its concept is similar to how our old graphing algorithm works since both of them expand the frontiers if the goal state is not yet found. Upon testing, we have found IDS is very inefficient and intensive because of the large number of states that can be generated. The second search algorithm we considered was A* Search. But before we implemented the search, we had to find out what is a suitable heuristic for Sokoban. We eventually arrived at two possible heuristics and these are the Manhattan Distance only and the other is the number of open goals. We tried to first implement Manhattan Distance only and it worked with a very obvious test case where the bot just has to move to the right in order to solve the problem. However, upon using it in other problems especially with very obvious maps with two boxes, it did not

work. A probable cause to A* Search not working is because it is optimal, which means that it is going to be very slow since it tries to find the most optimal solution instead of focusing on speed. We ended up with Greedy Best-First Search as our algorithm of choice because it is made for speed and we initially used Manhattan Distance only as the heuristic. After testing, the algorithm did solve some of the puzzles but we decided to change the heuristic again which is now the Manhattan Distance and the number of boxes not yet in their goal spot. The algorithm was slightly faster and we have decided to finalize that as our heuristic.

C. Implementation of the algorithm

Implementing the algorithm was complex because we had to create an algorithm that generates the legal states given the state that is being evaluated. The code for the generation of states went through several iterations and we had to create our own maps for very specific scenarios like a box being behind another box or a wall being behind a box just to be sure that the state will not be generated since the box cannot be pushed. Another challenge we encountered was that we had to use data structures that we have not used before like Hash Sets and Priority Queues since a lot of the resources we referred to used these data structures. Another one is that we had to change the heuristic since we had to come up with different heuristics several times during our design process. Lastly, testing and debugging was uneasy because when the program was running, we were not sure if the searching was actually still searching or was in an infinite loop. We also had difficulty in determining if there was a problem with the code of the searching algorithm or the state representation itself.

V. **Conclusion:**

There are several search algorithms that can be utilized to solve Sokoban but each of them have their own advantages and disadvantages. The group mostly considered two significant factors in implementing the efficient algorithm such as the speed and its optimality. Considering the bot has only 15 seconds to return a solution, our group decided to focus on speed. The group also noticed that the more crates a given map has, its time in seconds for a solution to be returned increases by a factor of the amount of crates and also with a factor of the map size.

Some realizations of the group was that using the most optimal algorithm is important to reaching the end goal of the project. Another insight is that when implementing the algorithms in the code, not all algorithms have the same solution time. Some of the algorithms performed better than others in certain maps and in certain situations. Although, the program/bot was able to perform most of the maps required by the project specifications.

Another realization of the group was about which heuristic calculations to use as using any informed search, either GBFS or A*, will perform differently using different heuristic computations. The group had found a report regarding their Sokoban solver in which they had analysis of different searches in which GBFS performs better with a Manhattan Distance computed heuristic while their A* performed better with the number of open goals heuristic. The group collectively ended up deciding that a combination of both heuristic values will lead to a more accurate heuristic value thus making the program choose a path that more closely aligns with the minimal cost path.

Ultimately the group learned that it is best to use informed searches to even remotely get close to getting a solution with a faster time limit. Different informed searches are slower or faster depending on which heuristic fits that informed search best. The group also learnt the differences between GBFS and A* where GBFS is made for speedier outputs while A* will lead to more optimized outputs.

VI. **Table of Contributions**:

Here is a table showing the contributions of each member to the project.

Name of Member	**Tasks/Contributions**
Annika Dominique S. Campos	Coder (Algorithm), Algorithm/Heuristic Designer, Wrote the Paper
Carl Vincent Blix Lingat	Leader, Coder (State Representation), Algorithm/Heuristic Designer, and Wrote the Paper
Daniel Gavrie Clemente	Tester and Wrote the Paper
Jewel Claire M.Isolan	Coder (Algorithm), Wrote the Paper
Rizza Chan	Algorithm/Heuristic Designer, Tester, and Wrote the Paper

VII. **References:**

1) Gao, H., Huang, X., Liu, S., Wang, G., & Zhong, Z. (2019, March 9). A Sokoban Solver Using Multiple Search Algorithms and Q-Learning. GitHub. https://github.com/XUHUAKing/sokoban-qlearning/tree/master

2) Postma, J. O. (2016). Generic Puzzle Level Generation for Deterministic Transportation Puzzles (Master's thesis).

3) Selawsky, J. (2019, March 9). A* Search Algorithm in Java. CodeGym. https://codegym.cc/groups/posts/a-search-algorithm-in-java?fbclid=IwAR3WiAoFAJHRbQRyoCsEgmlKzlZ-RexNOlluM2Q56sq-8PhZs90otzVYe9Y