SokoBot Project Report

Author: Evan Riley Calupig
Course: CSINTSY - Introduction to Intelligent Systems

---

1. SokoBot Algorithm

SokoBot solves Sokoban puzzles using a Greedy Best-First Search algorithm. It contains optimizations to search faster. The bot needed to find a solution in 15 seconds. Speed took precedence over finding the shortest path.

1.1. State Representation

A game state contains the puzzle's moving parts. The GameState class stores the player's location and all crate locations. It also stores the path taken and a unique hash for the state.

Excerpt from `solver/model/GameState.java` *:*

```
public final class GameState {
    /** The current position of the player on the game board. */
    public final Point player;

    /** A set containing the positions of all crates on the game board. */
    public final Set<Point> crates;

    /** A string representing the sequence of moves to reach this state. */
    public final String path;

    /** The Zobrist hash value for this specific game state. */
    public final long zobristHash;

    // ... constructor and other methods
}
```

A GameBoard object stores the fixed parts of the map, like walls and goals. The program creates this object once. It pre-computes data to speed up the search.

Excerpt from `solver/model/GameBoard.java` *:*

```
public final class GameBoard {
    private final char[][] board;
    private final int rows;
    private final int cols;
    private final Set<Point> goals;
    private final Set<Point> deadSquares;

    public GameBoard(char[][] board) {
        this.board = board;
        this.rows = board.length;
        this.cols = board[0].length;
        this.goals = findTiles(Constants.GOAL);
        // Pre-computes squares where a crate can never be pushed from.
        this.deadSquares = computeDeadSquares();
    }
    // ... other methods
}
```

1.2. Search Algorithm and Heuristic

The solver uses a Greedy Best-First Search. A PriorityQueue manages the states for exploration. The queue sorts states by their heuristic value. This guides the search toward a goal.

The main search loop takes the best state from the queue. Then it generates all possible next states from that state.

The main search loop from `SokoBot.java` *:*

```
public String solve(GameBoard board) {
    // ... initialization of openList and closedList ...

    while (!openList.isEmpty()) {
        GameState currentState = openList.poll();

        if (isGoal(currentState, board)) {
            return currentState.path; // Solution found
        }

        closedList.add(currentState.zobristHash);
        expandState(currentState, openList, closedList);
    }
    return "No solution found"; // Should not happen with valid puzzles
}
```

The heuristic function sums the Manhattan distances for the best crate and goal pairs. This calculation gives a quick estimate of the remaining work.

Excerpt from `solver/util/Heuristic.java` *:*

```
public int calculate(GameState state, GameBoard board) {
    int totalMinDistance = 0;
    Set<Point> availableGoals = new HashSet<>(board.getGoals());

    for (Point crate : state.getCrates()) {
        int minDistance = Integer.MAX_VALUE;
        Point bestGoal = null;

        for (Point goal : availableGoals) {
            int distance = manhattanDistance(crate, goal);
            if (distance < minDistance) {
                minDistance = distance;
                bestGoal = goal;
            }
        }
        totalMinDistance += minDistance;
        // Each goal can only be assigned to one crate.
        if (bestGoal != null) {
            availableGoals.remove(bestGoal);
        }
    }
    return totalMinDistance;
}
```

1.3. Key Optimizations

A simple Greedy Best-First Search is not enough to solve Sokoban. The bot's performance comes from two techniques that reduce the search space.

1. Zobrist Hashing: A closedList tracks visited states with 64-bit Zobrist hashes. A game state's hash is the XOR sum of random numbers for each piece's position. This method checks for past states very quickly.

The hashing system starts with a table of random numbers.

Excerpt from `solver/util/Zobrist.java` *:*

```
public final class Zobrist {
    private final long[][] table;

    public Zobrist(int rows, int cols) {
        // Two piece types: Player and Crate
        this.table = new long[rows * cols][2];
        Random rand = new Random();

        for (int i = 0; i < rows * cols; i++) {
            table[i][Constants.PLAYER_ZOBRIST_INDEX] = rand.nextLong();
            table[i][Constants.CRATE_ZOBRIST_INDEX] = rand.nextLong();
        }
    }
    // ... hashing methods
}
```

2. Multiple Deadlock Detectors: The largest optimization is an aggressive deadlock detector. The system checks a new state for unsolvable conditions before it is added to the open list.

Example: Simple Corner Deadlock check from `DeadlockDetector.java` *:*

```java
private boolean isSimpleCorner(Point crate, GameBoard board) {
    // A crate in a corner is a deadlock if it is not a goal square.
    if (board.getTile(crate) == Constants.GOAL) {
        return false;
    }

    boolean upWall = board.getTile(
        crate.getNeighbor(Direction.UP)) == Constants.WALL;
    boolean downWall = board.getTile(
        crate.getNeighbor(Direction.DOWN)) == Constants.WALL;
    boolean leftWall = board.getTile(
        crate.getNeighbor(Direction.LEFT)) == Constants.WALL;
    boolean rightWall = board.getTile(
        crate.getNeighbor(Direction.RIGHT)) == Constants.WALL;

    // Adjacent to two perpendicular walls
    return (upWall || downWall) && (leftWall || rightWall);
}
```

This check occurs inside the main search loop. It prunes bad search branches early.

Pruning logic in `SokoBot.java` *:*

```java
private boolean isStateViable(GameState state, Set<Long> closedList) {
    if (state == null || closedList.contains(state.zobristHash)) {
        return false;
    }
    // The call to the deadlock detector
    return !deadlockDetector.isDeadlock(state, board);
}
```

---

2. Evaluation and Performance

2.1. Strengths

SokoBot's main strength is its speed on constrained puzzles.

```
  #####
 #    @#
 #$ $#####
# .#   $  #
#*.. $    #
# *.*.  # #
#   ##$## #
 ### #    #
      ####
```

Effective on Constrained Maps: The bot performs well on maps with many hallways, corners, and small rooms. The deadlock detection finds and removes many unsolvable states on these maps.

Time: The greedy search, fast duplicate checking, and aggressive pruning help the bot find a solution within 15 seconds for its target difficulty.

Scalability: The optimizations mean the bot's performance scales well. Performance drops slowly as the number of crates increases. The bot spends less time on bad moves.

A puzzle with corner goals is easy for the bot. It knows that pushing a crate into a non-goal corner creates a deadlock, so it does not explore that path.

2.2. Weaknesses

The bot is less effective on wide-open puzzles. It also struggles with puzzles needing complex crate movements.

```
 ########################
 #  #  #  #  #  #  #  #  #
 # $#$ # $#$ # $#$ # $#$ #
 #  #  #  #  #  #  #  #  #
 ## # ### # ### # ### # ##
 #  #  #  #  #  #  #  #  #
 #     #     #     #     #
## ### # ### # ### # ### ##
#   #  #  #  #  #  #  #   #
#   #  #  #  @  #  #  #   #
##  # ### # ### # ### #  ##
 #  .  . . .   . . .  .  #
 ########################
```

Heuristic Traps: The Manhattan distance heuristic ignores walls and other crates. This can create traps. On open maps, the bot can move a crate to a spot numerically closer to a goal. This new spot can be harder to solve from.

Sub-Optimal Solutions: The search does not weigh the path cost. So the solutions it finds are often not the shortest. The bot prefers apparent progress, which can create longer paths.

Struggles with "Parking" Maneuvers: The bot has trouble with levels that require pushing a crate far from its goal to clear a path for another crate. Such moves increase the heuristic value. The greedy algorithm considers these moves last.

Puzzles with a large central room are hard for the bot. These puzzles require re-ordering crates. The deadlock detector finds few problems in these spaces. The simple heuristic does not provide a strong guide. The bot must explore a much larger state space.

---

3. Challenges

The main challenge was managing the large number of possible game states. The number of future states grows quickly with each move. A simple search is not possible. The project's focus shifted. It changed from implementing a search algorithm to building a state-pruning system that performs well.

Implementing the deadlock detectors was a large technical challenge. Corner deadlocks are simple. Creating a system to find more complex patterns required pre-computation. This process also required a deeper analysis of the game's rules.

Setting up the development environment was a personal challenge. I used the command line, Gradle, and Neovim from scratch. This setup took more initial work than a standard IDE. The result was a light, customized workflow. This workflow made development and testing easier.

---

4. Table of Contributions

Contribution	Contributor
Project setup with Gradle and directory structure	Evan Riley Calupig
Design and implementation of the core search algorithm (GBFS)	Evan Riley Calupig
State representation (`GameState` , `GameBoard` , `Point`)	Evan Riley Calupig
Implementation of the heuristic function (Manhattan Distance)	Evan Riley Calupig
Advanced optimization: Zobrist Hashing for state tracking	Evan Riley Calupig
Advanced optimization: Multi-level Deadlock Detection	Evan Riley Calupig
Integration of the solver with the provided Java Swing GUI	Evan Riley Calupig
Creation of test maps and performance evaluation	Evan Riley Calupig
Documentation (README.md and Project Report)	Evan Riley Calupig

---

5. Declaration of AI Usage

AI Tool(s) Used: Gemini 2.5 Pro

How AI Was Used: I used Gemini for brainstorming. Early in the project, I used it to learn the differences between search algorithms. The problem involved Sokoban with a time limit. Later, I used the AI to find advanced optimization techniques for game solvers.

Sample Prompt:

> *"I am working on a Sokoban solver project in Java for a university course. The main constraints are a 15-second time limit for the solver to find a solution. The test levels will have between 2 to 8 crates. I need to choose an appropriate search algorithm. Should I use A* search or something else, like Greedy Best-First Search? What are the trade-offs in the context of Sokoban and this time limit? Also, are there any common, advanced techniques for optimizing Sokoban solvers that I should be aware of to handle the large state space?"*

Sample AI Output Summary:

> *The AI explained that for the 15-second time limit, Greedy Best-First Search was a more practical choice than A*. It finds any solution quickly, not just the best one. It then introduced two important optimization techniques: **Zobrist hashing** for efficient duplicate state detection and the importance of implementing robust **deadlock detection** to prune unsolvable branches of the search tree. It mentioned common patterns like corner deadlocks.*

Reflection on Learning: Using the AI helped make this a more sophisticated and effective solver. The tool acted as a mentor. It introduced industry-standard techniques like Zobrist hashing. I did not know about this technique before. I then focused my learning on these advanced concepts. The project became a deeper and more rewarding educational experience.