# 1 fr.valax.sokoshell.solver

## 1.1 pathfinder

### PlayerAStar

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.PriorityQueue;

/**
 * An 'A*' that can find a path between a start position and an end position for a player.
 * It uses a local mark system.
 */
public class PlayerAStar extends AbstractAStar {

    private final int boardWidth;
    private final AStarMarkSystem markSystem;
    private final Node[] nodes;

    public PlayerAStar(Board board) {
        super(new PriorityQueue<>(board.getWidth() * board.getHeight()));
        this.boardWidth = board.getWidth();
        markSystem = new AStarMarkSystem(board.getWidth() * board.getHeight());
        nodes = new Node[board.getHeight() * board.getWidth()];

        for (int i = 0; i < nodes.length; i++) {
            nodes[i] = new Node();
        }
    }

    private int toIndex(TileInfo player) {
        return player.getY() * boardWidth + player.getX();
    }

    @Override
    protected void init() {
        markSystem.unmarkAll();
        queue.clear();
    }

    @Override
    protected void clean() {

    }

    @Override
    protected Node initialNode() {
        int i = toIndex(playerStart);

        Node init = nodes[i];
        init.setInitial(playerStart, null, heuristic(playerStart));
        return init;
    }

    @Override
    protected Node processMove(Node parent, Direction dir) {
        TileInfo player = parent.getPlayer();
        TileInfo dest = player.adjacent(dir);

        if (dest.isSolid()) {
            return null;
        }

        int i = toIndex(dest);
        Node node = nodes[i];

        if (markSystem.isMarked(i) || markSystem.isVisited(i)) { // the node was added to the queue, therefore node.getExpectedDist() is valid
            if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {
                node.changeParent(parent);
                decreasePriority(node);
            }

            return null;
        } else {
            markSystem.mark(i);
            node.set(parent, dest, null, heuristic(dest));
            return node;
        }
    }

    @Override
    protected void markVisited(Node node) {
        markSystem.setVisited(toIndex(node.getPlayer()));
    }

    @Override
    protected boolean isVisited(Node node) {
        return markSystem.isVisited(toIndex(node.getPlayer()));
    }

    protected int heuristic(TileInfo newPlayer) {
        return newPlayer.manhattanDistance(playerDest);
    }

    @Override
    protected boolean isEndNode(Node node) {
        return node.getPlayer().isAt(playerDest);
    }
}
```

### AStarMarkSystem

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.mark.Mark;
import fr.valax.sokoshell.solver.board.mark.MarkSystem;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

/**
 * A mark is visited, if it is equal to the global mark.
 * A mark is marked, if it is equal to the global mark minus one.
 * It is used because, in A*, I need to know when I first encounter
 * a node (mark) and when I poll a node from the PriorityQueue (visited).
 * A node which isn't marked has a wrong expected dist, inherited from a previous
 * call to {@link AbstractAStar#findPath(TileInfo, TileInfo, TileInfo, TileInfo)}
 */
public class AStarMarkSystem implements MarkSystem {

    private int mark = 0;
    private final AStarMark[] marks;

    public AStarMarkSystem(int capacity) {
        marks = new AStarMark[capacity];

        for (int i = 0; i < capacity; i++) {
            marks[i] = new AStarMark();
        }
    }

    @Override
    public Mark newMark() {
        throw new UnsupportedOperationException();
    }

    /**
     * Unmark and <strong>un-visit</strong> all mark
     */
    @Override
    public void unmarkAll() {
        mark += 2;
    }

    public void mark(int i) {
        marks[i].mark();
    }

    public void setVisited(int i) {
        marks[i].setVisited();
    }

    public boolean isMarked(int i) {
        return marks[i].isMarked();
    }

    public boolean isVisited(int i) {
        return marks[i].isVisited();
    }

    @Override
    public void reset() {
        mark = 0;

        for (AStarMark mark : marks) {
            mark.unmark();
        }
    }

    @Override
    public int getMark() {
        return 0;
    }

    private class AStarMark implements Mark {

        private int mark = AStarMarkSystem.this.mark - 2;

        @Override
        public void mark() {
            mark = AStarMarkSystem.this.mark - 1;
        }

```

```java
        public void markVisited() {
            mark = AStarMarkSystem.this.mark - 1;
        }

        public void setVisited() {
            mark = AStarMarkSystem.this.mark;
        }

        @Override
        public void unmark() {
            mark = AStarMarkSystem.this.mark - 2;
        }

        @Override
        public boolean isMarked() {
            return mark == AStarMarkSystem.this.mark - 1;
        }

        public boolean isVisited() {
            return mark == AStarMarkSystem.this.mark;
        }

        @Override
        public MarkSystem getMarkSystem() {
            return AStarMarkSystem.this;
        }
    }
}
```

### AbstractAStar

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.Move;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.PriorityQueue;

/**
 * Abstract implementation of A*.
 */
public abstract class AbstractAStar {

    protected TileInfo playerStart;
    protected TileInfo crateStart;
    protected TileInfo playerDest;
    protected TileInfo crateDest;

    protected final PriorityQueue<Node> queue;

    public AbstractAStar(PriorityQueue<Node> queue) {
        this.queue = queue;
    }

    /**
     * @return true if path exists
     * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
     */
    public boolean hasPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart, TileInfo crateDest) {
        return findPath(playerStart, playerDest, crateStart, crateDest) != null;
    }

    /**
     * It also computes the move field in {@link Node}
     *
     * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
     */
    public Node findPathAndComputeMoves(TileInfo playerStart, TileInfo playerDest,
    ↪   TileInfo crateStart, TileInfo crateDest) {
        Node end = findPath(playerStart, playerDest, crateStart, crateDest);

        if (end == null) {
            return null;
        }

        Node current = end;
        while (current.getParent() != null) {
            Node last = current.getParent();

            TileInfo lastPlayer = last.getPlayer();
            TileInfo currPlayer = current.getPlayer();
            Direction dir = Direction.of(currPlayer.getX() - lastPlayer.getX(),
            ↪   currPlayer.getY() - lastPlayer.getY());

            boolean moved = crateStart != null &&
            ↪   !current.getCrate().isAt(last.getCrate());
            current.setMove(Move.of(dir, moved));

            current = last;
        }

        return end;
    }

    /**
     * Find a path between (playerStart, crateStart) and (playerDest, crateDest).
     * The returned node may be cached by the implementation. Therefore, if you
     * want to keep the path in memory, you need to copy the path.
     *
     * @param playerStart player start
     * @param playerDest player dest
     * @param crateStart crate start
     * @param crateDest crate dest
     * @return the shortest path as a linked list in reverse.
     */
    public Node findPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart,
    ↪   TileInfo crateDest) {
        this.playerStart = playerStart;
        this.crateStart = crateStart;
        this.playerDest = playerDest;
        this.crateDest = crateDest;

        init();
        Node n = initialNode();
        queue.offer(n);

        // int c = 0;
        Node end = null;
        while (!queue.isEmpty()) {
            Node node = queue.poll();

            if (isEndNode(node)) {
```

```java
                end = node;
                break;
            }

            if (isVisited(node)) {
                continue;
            }

            for (Direction direction : Direction.VALUES) {
                Node child = processMove(node, direction);

                if (child != null) {
                    queue.offer(child);
                }
            }

            markVisited(node);
            // c++;
        }
        // System.out.println(c);

        clean();
        return end;
    }

    /**
     * Decrease the priority of the node in the queue if and only if it is in the queue
     * @param node node
     */
    public void decreasePriority(Node node) {
        // TODO: we do not have a fixed size binary heap that
        //  can efficiently decrease priority (at least O(log n))
        if (queue.remove(node)) { // takes O(n)
            queue.offer(node); // takes O(log n)
        }
    }

    /**
     * Init A*. Usually clear the queue. Called before the search
     */
    protected abstract void init();

    /**
     * Clean the object. Called at the end of the search
     */
    protected abstract void clean();

    /**
     * Returns the initial node.
     * @return the initial node
     */
    protected abstract Node initialNode();

    /**
     *
     * @param parent parent node
     * @param dir direction taken player
     * @return {@code null} if the player cannot move in the specified direction
     * or if the node was already visited. Otherwise, returns child node
     */
    protected abstract Node processMove(Node parent, Direction dir);

    /**
     * Mark the node as visited
     * @param node node
     */
    protected abstract void markVisited(Node node);

    /**
     * @param node node
     * @return {@code true} if the node is visited
     */
    protected abstract boolean isVisited(Node node);

    /**
     * @param node node
     * @return {@code true} if this node represents the solution
     */
    protected abstract boolean isEndNode(Node node);
}
```

### CratePlayerAStar

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

/**
 * Find the shortest path between (player start, crate start) and (player dest, crate dest):
 * the player moves a crate from 'crate start' to 'crate dest' and then moves to 'player dest'.
 */
public class CratePlayerAStar extends CrateAStar {

    public CratePlayerAStar(Board board) {
        super(board);
    }

    @Override
    protected boolean isEndNode(Node node) {
        return node.getPlayer().isAt(playerDest) && node.getCrate().isAt(crateDest);
    }

    @Override
    protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
         /*
            Try to first move the player near the crate
            Then push the crate to his destination
            Finally moves the player to his destination
         */
        int remaining = newCrate.manhattanDistance(crateDest);
        if (remaining == 0) {
            remaining = newPlayer.manhattanDistance(playerDest);
        } else {
            if (newPlayer.manhattanDistance(newCrate) > 1) {
                remaining += newPlayer.manhattanDistance(newCrate);
            }

            remaining += crateDest.manhattanDistance(playerDest);
        }

        return remaining;
    }
}
```

### Node

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.Move;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.Objects;

/**
 * A node in A*
 */
public class Node implements Comparable<Node> {

    private Node parent;
    private int dist;
    private int heuristic;
    private TileInfo player;
    private TileInfo crate;
    private Move move;

    private int expectedDist;

    public Node() {
    }

    public Node(Node parent,
                int dist, int heuristic,
                TileInfo player, TileInfo crate, Move move) {
        this.parent = parent;
        this.dist = dist;
        this.heuristic = heuristic;
        this.player = player;
        this.crate = crate;
        this.move = move;
    }

    public void setInitial(TileInfo player, TileInfo crate, int heuristic) {
        parent = null;
        dist = 0;
        this.heuristic = heuristic;
        this.player = player;
        this.crate = crate;

        expectedDist = heuristic;
    }

    public void set(Node parent, TileInfo player, TileInfo crate, int heuristic) {
        this.parent = parent;
        this.dist = parent.dist + 1;
        this.heuristic = heuristic;
        this.player = player;
        this.crate = crate;

        expectedDist = dist + heuristic;
    }

    public void changeParent(Node newParent) {
        this.parent = newParent;
        this.dist = newParent.dist + 1;

        expectedDist = dist + heuristic;
    }

    public Node getParent() {
        return parent;
    }

    public int getDist() {
        return dist;
    }

    public int getHeuristic() {
        return heuristic;
    }

    public TileInfo getPlayer() {
        return player;
    }

    public TileInfo getCrate() {
        return crate;
    }

    public Move getMove() {
        return move;
    }

    public void setMove(Move move) {
        this.move = move;
    }

    public int getExpectedDist() {
        return expectedDist;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Node node)) return false;

        if (!Objects.equals(player, node.player)) return false;
        return Objects.equals(crate, node.crate);
    }

    @Override
    public int hashCode() {
        int result = player != null ? player.getIndex() : 0;
        result = 31 * result + (crate != null ? crate.getIndex() : 0); // TODO
        return result;
    }

    @Override
    public int compareTo(Node o) {
        return Integer.compare(expectedDist, o.expectedDist);
    }
}
```

### CrateAStar

```java
package fr.valax.sokoshell.solver.pathfinder;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.PriorityQueue;

/**
 * Moves a crate from a start position to a destination.
 */
public class CrateAStar extends AbstractAStar {

    private final int boardWidth;
    private final int area;

    private final AStarMarkSystem markSystem;
    private final Node[] nodes;

    public CrateAStar(Board board) {
        super(new PriorityQueue<>(2 * board.getWidth() * board.getHeight()));
        this.boardWidth = board.getWidth();

        area = board.getWidth() * board.getHeight();
        markSystem = new AStarMarkSystem(area * area);

        nodes = new Node[area * area];

        for (int i = 0; i < nodes.length; i++) {
            nodes[i] = new Node();
        }
    }

    private int toIndex(TileInfo player, TileInfo crate) {
        return (player.getY() * boardWidth + player.getX()) * area + crate.getY() * boardWidth + crate.getX();
    }

    @Override
    protected void init() {
        markSystem.unmarkAll();
        queue.clear();
        crateStart.removeCrate();
    }

    @Override
    protected void clean() {
        crateStart.addCrate();
    }

    @Override
    protected Node initialNode() {
        int i = toIndex(playerStart, crateStart);

        Node init = nodes[i];
        init.setInitial(playerStart, crateStart, heuristic(playerStart, crateStart));
        return init;
    }

    @Override
    protected Node processMove(Node parent, Direction dir) {
        TileInfo player = parent.getPlayer();
        TileInfo crate = parent.getCrate();
        TileInfo playerDest = player.adjacent(dir);
        TileInfo crateDest = crate;

        if (playerDest.isAt(crate)) {
            crateDest = playerDest.adjacent(dir);

            if (crateDest.isSolid()) {
                return null;
            }

            // check deadlock
            if (!crateDest.isAt(this.crateDest) && // not a deadlock is if is destination
                    crateDest.adjacent(dir).isSolid() && // front must be solid
                    (crateDest.adjacent(dir.left()).isSolid() || // perp must be solid
                            crateDest.adjacent(dir.right()).isSolid())) {
                return null;
            }
        } else if (playerDest.isSolid()) {
            return null;
        }

        int i = toIndex(playerDest, crateDest);
        Node node = nodes[i];

        if (markSystem.isMarked(i) || markSystem.isVisited(i)) {
            if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {
                node.changeParent(parent);
                decreasePriority(node);
            }

            return null;
        } else {
            markSystem.mark(i);
            node.set(parent, playerDest, crateDest, heuristic(playerDest, crateDest));

            return node;
        }
    }

    @Override
    protected void markVisited(Node node) {
        markSystem.setVisited(toIndex(node.getPlayer(), node.getCrate()));
    }

    @Override
    protected boolean isVisited(Node node) {
        return markSystem.isVisited(toIndex(node.getPlayer(), node.getCrate()));
    }

    @Override
    protected boolean isEndNode(Node node) {
        return node.getCrate().isAt(crateDest);
    }

    protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
        int h = newCrate.manhattanDistance(crateDest);

        /* the player first need to move near the crate to push it
           may not be optimal for level like this:

            #########
            #       #
            # ##### #
            # ##### #
            # ##### #
             @$     # The player needs to do a detour to push the crate
            # #######
         */
        if (newPlayer.manhattanDistance(newCrate) > 1) {
            h += newPlayer.manhattanDistance(newCrate);
        }

        return h;
    }
}
```

## 1.2 collections

### Node

```
package fr.valax.sokoshell.solver.collections;

public class Node<E> {

    protected Node<E> next;
    protected E value;

    public Node(E value) {
        this.value = value;
    }

    /**
     * Detach this node from the linked list. After this call
     * {@link #next()} will return null. If any node has for next
     * this node, it won't be detached from these nodes.
     *
     * @return next node
     */
    public Node<E> detach() {
        Node<E> oldNext = next;
        next = null;
        return oldNext;
    }

    /**
     * Makes the specified node the previous node of this node.
     *
     * @param node new parent
     */
    public void attach(Node<E> node) {
        node.next = this;
    }

    public Node<E> next() {
        return next;
    }

    public E getValue() {
        return value;
    }

    public void setValue(E value) {
```

```java
        this.value = value;
    }
}
```

### SolverCollection

```java
package fr.valax.sokoshell.solver.collections;

import fr.valax.sokoshell.solver.State;

public interface SolverCollection<T extends State> {

    void clear();

    boolean isEmpty();

    int size();

    void addState(T state);

    T popState();

    T peekState();

    T peekAndCacheState();

    T cachedState();
}
```

### MinHeap

```java
package fr.valax.sokoshell.solver.collections;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MinHeap<T> {

    /**
     * Array of nodes.
     */
    protected final List<Node<T>> nodes;

    protected int currentSize;

    public MinHeap() {
        nodes = new ArrayList<>();
        currentSize = -1;
    }

    /**
     * Creates a min heap of fixed capacity.
     * This has 2 major consequences :
     * <ul>
     *     <li>this constructor instantiates empty object in each of the cases of the min heap array</li>
     *     <li>When {@link MinHeap#add(Object, int)} is called, no element is created nor added : the case where the
     *     new element goes is only updated with the new object values.</li>
     * </ul>
     * @param capacity The (fixed) capacity of the heap
     */
    public MinHeap(int capacity) {
        nodes = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            nodes.add(i, new Node<T>());
        }
        currentSize = 0;
    }

    protected int leftChild(int i) {
        return 2 * i + 1;
    }

    protected int rightChild(int i) {
        return 2 * i + 2;
    }

    protected void moveNodeUp(int i) {
        if (i == 0) {
            return;
        }
        final int p = parent(i);
        if (nodes.get(i).hasPriorityOver(nodes.get(p))) {
            Collections.swap(nodes, i, p);
            moveNodeUp(p);
        }
    }

    protected void moveNodeDown(int i) {
        int j = i;
        final int l = leftChild(i), r = rightChild(i);
        if (l < size() && nodes.get(l).hasPriorityOver(nodes.get(i))) {
            j = l;
        }
        if (r < size() && nodes.get(r).hasPriorityOver(nodes.get(l))) {
            j = r;
        }

        if (i != j) {
            Collections.swap(nodes, i, j);
            moveNodeDown(j);
        }
    }

    private int parent(int i) {
        assert i != 0;
        return (i - 1) / 2;
    }

    public void add(T content, int priority) {
        int i = 0;
        if (currentSize == -1) {
            nodes.add(new Node<>(content, priority));
            moveNodeUp(nodes.size() - 1);
        } else {
            nodes.get(currentSize).set(content, priority);
            moveNodeUp(currentSize);
            currentSize++;
        }
    }

    public T pop() {
        final int i = size() - 1;
        Collections.swap(nodes, 0, i);
        T content;
        if (currentSize == -1) {
            content = nodes.remove(i).content();
        } else {
            content = nodes.get(i).content();
            currentSize--;
        }
        moveNodeDown(0);
        return content;
    }

    public T peek() {
        return nodes.get(0).content();
    }

    public void clear() {
        if (currentSize == - 1) {
            nodes.clear();
        } else {
            currentSize = 0;
        }
    }

    public boolean isEmpty() {
        return currentSize == -1 ? nodes.isEmpty() : (currentSize == 0);
    }

    public int size() {
        return currentSize == -1 ? nodes.size() : currentSize;
    }

    /**
     * Min heap (state, priority) couple.
     */
    protected static final class Node<T> {
        private T content;
        private int priority;

        public Node() {
            set(null, Integer.MAX_VALUE);
        }

        public Node(T content, int priority) {
            set(content, priority);
        }

        public boolean hasPriorityOver(Node<T> o) {
            return priority < o.priority;
        }

        @Override
        public String toString() {
            return String.format("Node[priority=%d]", priority);
        }

        public void set(T content, int priority) {
            this.content = content;
            this.priority = priority;
        }

        public T content() {
            return content;
        }

        public void setContent(T content) {
            this.content = content;
        }

        public int priority() {
            return priority;
        }

        public void setPriority(int priority) {
            this.priority = priority;
        }
    }
}
```

### SolverPriorityQueue

```
package fr.valax.sokoshell.solver.collections;

import fr.valax.sokoshell.solver.WeightedState;

/**
 * Priority queue of dynamic capacity. The priority are in <strong>ASCENDANT</strong>
↪  order, i.e. the element returned
 * by {@link SolverPriorityQueue#popState()} with the <strong>LOWEST</strong> priority.
 */
public class SolverPriorityQueue implements SolverCollection<WeightedState> {

    /**
     * @implNote We use a min heap collection.
      */
    private final MinHeap<WeightedState> heap = new MinHeap<>();

    private WeightedState cachedState;

    @Override
    public void addState(WeightedState state) {
        heap.add(state, state.weight());
    }

    @Override
    public WeightedState popState() {
        return heap.pop();
    }

    @Override
    public WeightedState peekState() {
        return heap.peek();
    }

    @Override
    public WeightedState peekAndCacheState() {
        cachedState = popState();
        return cachedState;
    }

    @Override
    public WeightedState cachedState() {
```

```java
        return cachedState;
    }

    @Override
    public void clear() {
        heap.clear();
    }

    @Override
    public boolean isEmpty() {
        return heap.isEmpty();
    }

    @Override
    public int size() {
        return heap.size();
    }
}

```

## 1.3 heuristic

### AbstractHeuristic

```java
package fr.valax.sokoshell.solver.heuristic;

import fr.valax.sokoshell.solver.board.Board;

/**
 * Base class for heuristic computing classes.
 * As there are different ways to compute the heuristic of a state, we provide a set of
↪  class each implementing
 * different heuristic calculation methods.
 */
public abstract class AbstractHeuristic implements Heuristic {

    protected final Board board;

    public AbstractHeuristic(Board board) {
        this.board = board;
    }
}
```

### GreedyHeuristic

```java
package fr.valax.sokoshell.solver.heuristic;

import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

/**
 * According to <a
↪  href="http://sokobano.de/wiki/index.php?title=Solver#Greedy_approach">this article</a>
 */
public class GreedyHeuristic extends AbstractHeuristic {

    private final LinkedList list;

    public GreedyHeuristic(Board board) {
        super(board);
        final int n = board.getTargetCount();

        list = new LinkedList(n);
    }

    @Override
    public int compute(State s) {
        int heuristic = 0;

        board.getMarkSystem().unmarkAll();

        int n = 0;
        for (int crate : s.cratesIndices()) {
            TileInfo tile = board.getAt(crate);

            if (tile.isCrateOnTarget()) {
                tile.mark();
            } else {
                list.add(tile);

                n++;
            }
        }


        for (int i = 0; i < n; i++) {
            Node minNode = list.getHead();
            TileInfo.TargetRemoteness minDist = minNode.getNearestNotAttributedTarget();

            Node node = minNode.nextNode();
            while (node != null) {
                TileInfo.TargetRemoteness nearest = node.getNearestNotAttributedTarget();

                if (nearest.distance() < minDist.distance()) {
                    minNode = node;
                    minDist = nearest;
                }

                node = node.nextNode();
            }

            board.getAt(minDist.index()).mark();
            minNode.getCrate().mark();
            heuristic += minDist.distance();

            minNode.remove();
        }

        return heuristic;
    }

    private static class LinkedList {

        private final Node[] nodeCache;
        private int size = 0;

        private Node head;

        public LinkedList(int size) {
            nodeCache = new Node[size];

            for (int i = 0; i < size; i++) {
                nodeCache[i] = new Node(this);
            }
        }

        public void add(TileInfo crate) {
            Node newHead = nodeCache[size];
            newHead.set(crate);

            if (head != null) {
                newHead.next = head;
                head.previous = newHead;
            }
            head = newHead;

            size++;
        }

        public void remove(Node node) {
            if (node == head) {
                head = node.next;

                if (head != null) {
                    head.previous = null;
                }
            } else {
                node.previous.next = node.next;

                if (node.next != null) {
                    node.next.previous = node.previous;
                }
            }

            size--;
        }

        public Node getHead() {
            return head;
        }
    }

    private static class Node {

        private final LinkedList list;
        private TileInfo crate;

        private Node previous;
        private Node next;

        /**
         * Index in crate's target remoteness
         */
        private int index = 0;

        public Node(LinkedList list) {
            this.list = list;
        }

        public void set(TileInfo tile) {
            crate = tile;
            index = 0;
        }

        public void remove() {
```

```java
            list.remove(this);
        }

        public Node nextNode() {
            return next;
        }

        public TileInfo getCrate() {
            return crate;
        }

        public TileInfo.TargetRemoteness getNearestNotAttributedTarget() {
            TileInfo.TargetRemoteness[] remoteness = crate.getTargets();

            Board b = crate.getBoard();
            while (b.getAt(remoteness[index].index()).isMarked()) {
                index++;
            }

            return remoteness[index];
        }
    }
}
```

### SimpleHeuristic

```java
package fr.valax.sokoshell.solver.heuristic;

import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.Board;

/**
 * According to <a
↪   href="http://sokobano.de/wiki/index.php?title=Solver#Simple_Lower_Bound">this
↪   article</a>
 */
public class SimpleHeuristic extends AbstractHeuristic {

    public SimpleHeuristic(Board board) {
        super(board);
    }

    /**
     * Sums the distances to the nearest goal of each of the crates of the state.
     */
    public int compute(State s) {
        int h = 0;
        for (int i : s.cratesIndices()) {
            h += board.getAt(i).getNearestTarget().distance();
        }
        return h;
    }
}
```

### Heuristic

```java
package fr.valax.sokoshell.solver.heuristic;

import fr.valax.sokoshell.solver.State;

/**
```

```
 * Heuristic computing class for guided-search (e.g. A*)
 */
public interface Heuristic {

    /**
     * Computes the heuristic of the given state.
     * @param s the state to compute the heuristic
     * @return the heuristic of the state
     */
    int compute(State s);

}
```

## 1.4 board

### 1.4.1 tiles

**MutableTileInfo**

```
package fr.valax.sokoshell.solver.board.tiles;

import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.MutableBoard;
import fr.valax.sokoshell.solver.board.Room;
import fr.valax.sokoshell.solver.board.Tunnel;
import fr.valax.sokoshell.solver.board.mark.Mark;

/**
 * Mutable implementation of {@link TileInfo}.
 *
 * This class extends {@link GenericTileInfo} and implements the setters methods defined
↪  in
 * {@link TileInfo}.
 * It also implements getters and setters for the 'solver-intended' properties.
 *
 * @see TileInfo
 * @see GenericTileInfo
 */
public class MutableTileInfo extends GenericTileInfo {

    private final MutableBoard board;

    // Static information
    protected boolean deadTile;

    /**
     * The tunnel in which this tile is. A Tile is either in a room or in a tunnel
     */
    protected Tunnel tunnel;
    // contains for each direction, where is the outside of the tunnel from this tile
    protected Tunnel.Exit tunnelExit;
    protected Room room;

    /**
     * Remoteness data from this tile to every target on the board.
     */
    protected TargetRemoteness[] targets;

    /**
     * Nearest target on the board.
     */
    protected TargetRemoteness nearestTarget;

```

```java
    /**
     * The index of this crate in the {@link State#cratesIndices()} array
     */
    protected int crateIndex;


    // Dynamic information
    protected Mark reachable;
    protected Mark mark;

    public MutableTileInfo(MutableBoard board, Tile tile, int x, int y) {
        super(board, tile, x, y);
        this.board = board;

        this.reachable = board.getReachableMarkSystem().newMark();
        this.mark = board.getMarkSystem().newMark();
    }

    public MutableTileInfo(MutableBoard board, TileInfo other) {
        super(board, other);
        this.board = board;

        this.reachable = board.getReachableMarkSystem().newMark();
        this.mark = board.getMarkSystem().newMark();
    }

    // GETTERS //

    @Override
    public boolean isDeadTile() {
        return deadTile;
    }

    @Override
    public boolean isReachable() {
        return !tile.isSolid() && board.getCorral(this).containsPlayer();
    }

     @Override
    public Tunnel getTunnel() {
        return tunnel;
    }

    @Override
    public Tunnel.Exit getTunnelExit() {
        return tunnelExit;
    }

    public boolean isInATunnel() {
        return tunnel != null;
    }

    @Override
    public Room getRoom() {
        return room;
    }

    @Override
    public boolean isInARoom() {
        return room != null;
    }

    @Override
    public boolean isMarked() {
        return mark.isMarked();
    }

    @Override
    public TargetRemoteness getNearestTarget() {
        return nearestTarget;
    }

    @Override
    public TargetRemoteness[] getTargets() {
        return targets;
    }


    // SETTERS //

    @Override
    public void addCrate() {
        if (tile == Tile.FLOOR) {
            tile = Tile.CRATE;
        } else if (tile == Tile.TARGET) {
            tile = Tile.CRATE_ON_TARGET;
        }
    }

    @Override
    public void removeCrate() {
        if (tile == Tile.CRATE) {
            tile = Tile.FLOOR;
        } else if (tile == Tile.CRATE_ON_TARGET) {
            tile = Tile.TARGET;
        }
    }

    @Override
    public void setTile(Tile tile) {
        this.tile = tile;
    }

    @Override
    public void setDeadTile(boolean deadTile) {
        this.deadTile = deadTile;
    }

    @Override
    public void setReachable(boolean reachable) {
        this.reachable.setMarked(reachable);
    }

    @Override
    public void setTunnel(Tunnel tunnel) {
        this.tunnel = tunnel;
    }

    @Override
    public void setTunnelExit(Tunnel.Exit tunnelExit) {
        this.tunnelExit = tunnelExit;
    }

    @Override
```

```java
    public void setRoom(Room room) {
        this.room = room;
    }

    @Override
    public void mark() {
        mark.mark();
    }

    @Override
    public void unmark() {
        mark.unmark();
    }

    @Override
    public void setMarked(boolean marked) {
        mark.setMarked(marked);
    }

    @Override
    public void setTargets(TargetRemoteness[] targets) {
        this.targets = targets;
    }

    @Override
    public void setNearestTarget(TargetRemoteness nearestTarget) {
        this.nearestTarget = nearestTarget;
    }

    @Override
    public int getCrateIndex() {
        return crateIndex;
    }

    @Override
    public void setCrateIndex(int crateIndex) {
        this.crateIndex = crateIndex;
    }
}
```

### GenericTileInfo

```java
package fr.valax.sokoshell.solver.board.tiles;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Room;
import fr.valax.sokoshell.solver.board.Tunnel;

/**
 * A {@code package-private} class meant to be use as a base class for {@link TileInfo} implementations.
 * It defines all the basic properties and their corresponding getters
 * (position, tile, board, etc.)
 *
 * @see TileInfo
 */
public abstract class GenericTileInfo implements TileInfo {

    protected final Board board;

    protected final int x;
    protected final int y;

    protected Tile tile;


    /**
     * Create a new TileInfo
     *
     * @param tile the tile
     * @param x the position on the x-axis in the board
     * @param y the position on the y-axis in the board
     */
    public GenericTileInfo(Board board, Tile tile, int x, int y) {
        this.board = board;
        this.tile = tile;
        this.x = x;
        this.y = y;
    }

    public GenericTileInfo(TileInfo tileInfo) {
        this(tileInfo.getBoard(), tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
    }

    public GenericTileInfo(Board board, TileInfo tileInfo) {
        this(board, tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
    }

    @Override
    public Tile getTile() {
        return tile;
    }

    @Override
    public int getX() {
        return x;
    }

    @Override
    public int getY() {
        return y;
    }

    /**
     * Returns the board in which this tile is
     *
     * @return the board in which this tile is
     */
    public Board getBoard() {
        return board;
    }

    // SETTERS: throw UnsupportedOperationException as this class is immutable //

    @Override
    public void addCrate() {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void removeCrate() {
        throw new UnsupportedOperationException("Immutable object");
    }

```

```java
    @Override
    public void setTile(Tile tile) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setDeadTile(boolean deadTile) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setReachable(boolean reachable) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setTunnel(Tunnel tunnel) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setTunnelExit(Tunnel.Exit tunnelExit) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setRoom(Room room) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void mark() {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void unmark() {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setMarked(boolean marked) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setTargets(TargetRemoteness[] targets) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setNearestTarget(TargetRemoteness nearestTarget) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public void setCrateIndex(int index) {
        throw new UnsupportedOperationException("Immutable object");
    }

    @Override
    public int hashCode() {
        return y * board.getWidth() + x;
    }
}
```

### ImmutableTileInfo

```
package fr.valax.sokoshell.solver.board.tiles;

import fr.valax.sokoshell.solver.board.ImmutableBoard;
import fr.valax.sokoshell.solver.board.Room;
import fr.valax.sokoshell.solver.board.Tunnel;

/**
 * Immutable implementation of {@link TileInfo}.
 *
 * This class basically extends {@link GenericTileInfo}. It implements the setters methods defined in
 * {@link TileInfo} by throwing an {@link UnsupportedOperationException}.
 * It also implements the 'solver-intended' properties by always returning the default value: for instance, a
 * {@link ImmutableTileInfo} is never a 'dead tile', so the {@link #isDeadTile} method will always return {@code false}.
 * The same policy is applied for each property.
 *
 * @see TileInfo
 * @see GenericTileInfo
 */
public class ImmutableTileInfo extends GenericTileInfo {

    public ImmutableTileInfo(ImmutableBoard board, Tile tile, int x, int y) {
        super(board, tile, x, y);
    }

    public ImmutableTileInfo(TileInfo tileInfo) {
        super(tileInfo);
    }

    // GETTERS //

    @Override
    public boolean isDeadTile() {
        return false;
    }

    @Override
    public boolean isReachable() {
        return true;
    }

    @Override
    public Tunnel getTunnel() {
        return null;
    }

    @Override
    public Tunnel.Exit getTunnelExit() {
        return null;
    }

    @Override
    public boolean isInATunnel() {
        return false;
```

```java
    }

    @Override
    public Room getRoom() {
        return null;
    }

    @Override
    public boolean isInARoom() {
        return false;
    }

    @Override
    public boolean isMarked() {
        return false;
    }

    @Override
    public String toString() {
        return tile.toString();
    }

    @Override
    public TargetRemoteness getNearestTarget() {
        return null;
    }

    @Override
    public TargetRemoteness[] getTargets() {
        return null;
    }

    @Override
    public int getCrateIndex() {
        return -1;
    }
}
```

### TileInfo

```java
package fr.valax.sokoshell.solver.board.tiles;

import fr.valax.sokoshell.solver.Corral;
import fr.valax.sokoshell.solver.board.*;
import fr.valax.sokoshell.solver.board.mark.Mark;
import fr.valax.sokoshell.solver.board.mark.MarkSystem;

import java.util.List;

/**
 * The {@link TileInfo} interface defines the methods that {@link Board} implementations
↪  need to manage tiles,
 * for instance:
 * <ul>
 *     <li>the position</li>
 *     <li>the {@link Tile}</li>
 * </ul>
 * It defines a set of high-level interactions functions.
 *
 * @see Board
 */
public interface TileInfo {

    // GETTERS //

    /**
     * @return the position of this TileInfo on the x-axis
     */
    int getX();

    /**
     * @return the position of this TileInfo on the y-axis
     */
    int getY();

    /**
     * @return which tile is this TileInfo
     */
    Tile getTile();

    /**
     * @return true if there is a crate at this position
     */
    default boolean anyCrate() {
        return getTile().isCrate();
    }

    /**
     * @return true if there is a wall or a crate at this position
     */
    default boolean isSolid() {
        return getTile().isSolid();
    }

    /**
     * @return true if this TileInfo is exactly a floor
     */
    default boolean isFloor() {
        return getTile() == Tile.FLOOR;
    }

    /**
     * @return true if this TileInfo is exactly a wall
     */
    default boolean isWall() {
        return getTile() == Tile.WALL;
    }

    /**
     * @return true if this TileInfo is exactly a target
     */
    default boolean isTarget() {
        return getTile() == Tile.TARGET;
    }

    /**
     * @return true if this TileInfo is exactly a crate
     * @see #anyCrate()
     */
    default boolean isCrate() {
        return getTile() == Tile.CRATE;
    }

    /**
```

```java
     * @return true if this TileInfo is exactly a crate on target
     * @see #anyCrate()
     */
    default boolean isCrateOnTarget() {
        return getTile() == Tile.CRATE_ON_TARGET;
    }

    /**
     * Returns {@code true} if this tile is at the same position as 'other'
     * @param other other tile
     * @return {@code true} if this tile is at the same position as 'other'
     */
    default boolean isAt(TileInfo other) {
        return isAt(other.getX(), other.getY());
    }

    /**
     * Returns {@code true} if this tile is at the position (x; y)
     * @param x x location
     * @param y y location
     * @return {@code true} if this tile is at the position (x; y)}
     */
    default boolean isAt(int x, int y) {
        return x == getX() && y == getY();
    }

    /**
     * Returns the direction between this tile and other.
     *
     * @param other 'other' tile
     * @return the direction between this tile and other
     */
    default Direction direction(TileInfo other) {
        return Direction.of(other.getX() - getX(), other.getY() - getY());
    }

    /**
     * Returns the distance of manhattan between this tile and other
     *
     * @param other 'other' tile
     * @return the distance of manhattan between this tile and other
     */
    default int manhattanDistance(TileInfo other) {
        return Math.abs(getX() - other.getX()) + Math.abs(getY() - other.getY());
    }

    /**
     * @return {@code true} if this tile is a dead tile
     * @see MutableBoard#computeDeadTiles()
     */
    boolean isDeadTile();

    /**
     * @return {@code true} if this tile is reachable by the player.
     * @see MutableBoard#findReachableCases(int)
     */
    boolean isReachable();

    /**
     * Returns the tunnel in which this tile is
     *
     * @return the tunnel in which this tile is
     */
    Tunnel getTunnel();

    /**
     * Returns the {@link Tunnel.Exit} object associated with this tile info.
     * If the tile isn't in a tunnel, it returns null
     *
     * @return the {@link Tunnel.Exit} object associated with this tile info or {@code null
     * @see Tunnel.Exit
     */
    Tunnel.Exit getTunnelExit();

    /**
     * Returns {@code true} if this tile info is in a tunnel
     *
     * @return {@code true} if this tile info is in a tunnel
     */
    boolean isInATunnel();

    /**
     * Returns the room in which this tile is
     *
     * @return the room in which this tile is
     */
    Room getRoom();

    /**
     * Returns {@code true} if this tile info is in a room
     *
     * @return {@code true} if this tile info is in a room
     */
    boolean isInARoom();

    /**
     * @return {@code true} if this tile is marked
     * @see Mark
     * @see MarkSystem
     */
    boolean isMarked();

    /**
     * @param dir the direction
     * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
     * @throws IndexOutOfBoundsException if this TileInfo is near the border of the board and
     * the direction point outside the board
     */
    default TileInfo adjacent(Direction dir) {
        return getBoard().getAt(getX() + dir.dirX(), getY() + dir.dirY());
    }

    /**
     * @param dir the direction
     * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
     * or {@code null} if the adjacent tile is outside the board
     */
    default TileInfo safeAdjacent(Direction dir) {
        return getBoard().safeGetAt(getX() + dir.dirX(), getY() + dir.dirY());
    }

    /**
```

```
     * Returns the board in which this tile is
     *
     * @return the board in which this tile is
     */
    Board getBoard();

    default int getIndex() {
        return getY() * getBoard().getWidth() + getX();
    }

    /**
     * Represents the index of this crate in {@link
↪   fr.valax.sokoshell.solver.State#cratesIndices()}
     * array.
     * @return -1 if not set or the index of this crate in
     *         {@link fr.valax.sokoshell.solver.State#cratesIndices()} array.
     */
    int getCrateIndex();

    TargetRemoteness getNearestTarget();

    TargetRemoteness[] getTargets();

    /**
     * @implNote If you replace index by TileInfo, you will need to modify
↪   MutableBoard#StaticTile.
     * If you are too lazy to do that, create an issue on github
     */
    record TargetRemoteness(int index, int distance) implements
    ↪   Comparable<TargetRemoteness> {

        @Override
        public int compareTo(TargetRemoteness other) {
            return this.distance - other.distance;
        }

        @Override
        public String toString() {
            return "TR[d=" + distance + ", i=" + index + "]";
        }
    }


    // SETTERS //

    /**
     * If this was a floor, this is now a crate
     * If this was a target, this is now a crate on target
     * @throws UnsupportedOperationException if the {@code addCrate} operation isn't
     * supported by this TileInfo
     */
    void addCrate();

    /**
     * If this was a crate, this is now a floor
     * If this was a crate on target, this is now a target
     * @throws UnsupportedOperationException if the {@code removeCrate} operation isn't
     * supported by this TileInfo
     */
    void removeCrate();

    /**
```

```java
     * Sets the tile.
     * @param tile the new tile
     * @throws UnsupportedOperationException if the {@code setTile} operation isn't
     * supported by this TileInfo
     */
    void setTile(Tile tile);

    /**
     * Sets this tile as a dead tile or not
     * @throws UnsupportedOperationException if the {@code setDeadTile} operation isn't
     * supported by this TileInfo
     * @see MutableBoard#computeDeadTiles()
     */
    void setDeadTile(boolean deadTile);

    /**
     * Sets this tile as reachable or not by the player. It doesn't check if it's possible.
     * @throws UnsupportedOperationException if the {@code setReachable} operation isn't
     * supported by this TileInfo
     * @see MutableBoard#findReachableCases(int)
     */
    void setReachable(boolean reachable);

    /**
     * Sets the tunnel in which this tile is
     * @throws UnsupportedOperationException if the {@code setTunnel} operation isn't
     * supported by this TileInfo
     */
    void setTunnel(Tunnel tunnel);

    /**
     * Sets the {@link Tunnel.Exit} object associated with this tile info
     * @throws UnsupportedOperationException if the {@code setTunnelExit} operation isn't
     * supported by this TileInfo
     * @see Tunnel.Exit
     */
    void setTunnelExit(Tunnel.Exit tunnelExit);

    /**
     * Sets the room in which this tile is
     * @throws UnsupportedOperationException if the {@code setRoom} operation isn't
     * supported by this TileInfo
     */
    void setRoom(Room room);

    /**
     * Sets this tile as marked
     * @throws UnsupportedOperationException if the {@code mark} operation isn't
     * supported by this TileInfo
     * @see Mark
     * @see MarkSystem
     */
    void mark();

    /**
     * Sets this tile as unmarked
     * @throws UnsupportedOperationException if the {@code unmark} operation isn't
     * supported by this TileInfo
     * @see Mark
     * @see MarkSystem
     */
    void unmark();

    /**
     * Sets this tile as marked or not
     * @throws UnsupportedOperationException if the {@code setMarked} operation isn't
     * supported by this TileInfo
     * @see Mark
     * @see MarkSystem
     */
    void setMarked(boolean marked);

    /**
     * Set the distance to every targets
     * @param targets distance to every targets
     * @throws UnsupportedOperationException if the {@code setTargets} operation isn't
     * supported by this TileInfo
     */
    void setTargets(TargetRemoteness[] targets);

    /**
     * Set the nearest target
     * @param nearestTarget nearest target
     * @throws UnsupportedOperationException if the {@code setNearestTarget} operation
↪   isn't
     * supported by this TileInfo
     */
    void setNearestTarget(TargetRemoteness nearestTarget);

    /**
     * @see #getCrateIndex()
     */
    void setCrateIndex(int index);
}
```

### Tile

```
package fr.valax.sokoshell.solver.board.tiles;

/**
 * Represents the content of a case of the board.
 */
public enum Tile {

    FLOOR(false, false),
    WALL(true, false),
    CRATE(true, true),
    CRATE_ON_TARGET(true, true),
    TARGET(false, false);


    private final boolean solid;

    private final boolean crate;

    Tile(boolean solid, boolean crate) {
        this.solid = solid;
        this.crate = crate;
    }

    /**
     * Tells whether objects (i.e. player or crates) can move through the case or not.
     */
    public boolean isSolid() {
        return solid;
    }

    /**
     * Tells whether the case is occupied by a crate (on a target or not) or not.
     */
    public boolean isCrate() {
        return crate;
    }
}
```

### 1.4.2 mark

**HeavyweightMarkSystem**

```
package fr.valax.sokoshell.solver.board.mark;

import java.util.ArrayList;
import java.util.List;

/**
 * A heavyweight mark system contains a pointer to every mark associated with this system
 */
public class HeavyweightMarkSystem extends AbstractMarkSystem {

    protected final List<Mark> marks;

    public HeavyweightMarkSystem() {
        marks = new ArrayList<>();
    }

    @Override
    public Mark newMark() {
        Mark m = super.newMark();
        marks.add(m);

        return m;
    }

    @Override
    public void reset() {
        mark = 0;

        for (Mark m : marks) {
            m.unmark();
        }
    }
}
```

**FixedSizeMarkSystem**

```
package fr.valax.sokoshell.solver.board.mark;

public class FixedSizeMarkSystem implements MarkSystem {

    protected final FMark[] marks;
    protected int mark;

    public FixedSizeMarkSystem(int capacity) {
        marks = new FMark[capacity];
        for (int i = 0; i < capacity; i++) {
            marks[i] = new FMark();
        }
    }

    public void mark(int i) {
        marks[i].mark();
    }

    public boolean isMarked(int i) {
        return marks[i].isMarked();
    }

    @Override
    public Mark newMark() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void unmarkAll() {
        mark++;

        if (mark == 0) {
            reset();
        }
    }

    @Override
    public void reset() {
        mark = 0;

        for (FMark mark : marks) {
            mark.unmark();
        }
    }

    @Override
    public int getMark() {
        return mark;
    }

    private class FMark implements Mark {

        private int mark = 0;

        @Override
        public void mark() {
            mark = FixedSizeMarkSystem.this.mark;
        }

        @Override
        public void unmark() {
            mark = FixedSizeMarkSystem.this.mark - 1;
        }

        @Override
        public boolean isMarked() {
            return mark == FixedSizeMarkSystem.this.mark;
        }

        @Override
        public MarkSystem getMarkSystem() {
            return FixedSizeMarkSystem.this;
        }
    }
}
```

### Mark

```
package fr.valax.sokoshell.solver.board.mark;

/**
 * @see MarkSystem
 * @author PoulpoGaz
 */
public interface Mark {

    /**
     * Marks the object. After this method is called, {@link #isMarked()}
     * will return {@code true}
     */
    void mark();

    /**
     * Un-marks the object. After this method is called, {@link #isMarked()}
     * will return {@code false}
     */
    void unmark();

    /**
     * Mark or not the object. After this method is called, {@link #isMarked()}
     * will return {@code marked}
     */
    default void setMarked(boolean marked) {
        if (marked) {
            mark();
        } else {
            unmark();
        }
    }

    /**
     * @return true is the object is marked
     */
    boolean isMarked();

    /**
     * @return the {@link MarkSystem} associated with this mark
     */
    MarkSystem getMarkSystem();
}
```

### DefaultMark

```
package fr.valax.sokoshell.solver.board.mark;

public class DefaultMark implements Mark {

    private final MarkSystem markSystem;
    private int mark;

    public DefaultMark(MarkSystem markSystem) {
        this.markSystem = markSystem;
        unmark();
```

```java
    }

    @Override
    public void mark() {
        mark = markSystem.getMark();
    }

    @Override
    public void unmark() {
        mark = markSystem.getMark() - 1;
    }

    @Override
    public boolean isMarked() {
        return mark == markSystem.getMark();
    }

    @Override
    public MarkSystem getMarkSystem() {
        return markSystem;
    }
}
```

### MarkSystem

```java
package fr.valax.sokoshell.solver.board.mark;

/**
 * <p>
 *     A MarkSystem is used by dfs/bfs/others algorithm to avoid checking twice an object.
 *     With a MarkSystem, you don't need to unmark all visited objects
 *     {@link Mark} associated with this system can be created using {@link #newMark()}.
 * </p>
 * <h2>How it works</h2>
 * <p>
 *     A mark have a value, the same for a MarkSystem. A mark is marked if it value is
↪   equals
 *     to the value of the MarkSystem. So, to unmark all mark, you just have to increase
 *     the MarkSystem's value.
 * </p>
 * @see Mark
 * @author PoulpoGaz
 */
public interface MarkSystem {

    /**
     * Create a new mark associated with this MarkSystem.
     * The mark is by default unmarked
     * @return a new mark
     */
    Mark newMark();

    /**
     * Unmark all marks
     */
    void unmarkAll();

    /**
     * Set the 'selected' mark to 0 and unmark all Mark
     */
    void reset();

    /**
     * @return the selected mark.
     */
    int getMark();
}
```

### AbstractMarkSystem

```java
package fr.valax.sokoshell.solver.board.mark;

/**
 * Contains the basic for all mark system
 */
public abstract class AbstractMarkSystem implements MarkSystem {

    /**
     * A mark is marked if it's value is equals to this field
     */
    protected int mark;

    @Override
    public Mark newMark() {
        return new DefaultMark(this);
    }

    @Override
    public void unmarkAll() {
        mark++;

        if (mark == 0) {
            reset();
        }
    }

    @Override
    public abstract void reset();

    @Override
    public int getMark() {
        return mark;
    }
}
```

### Tunnel

```java
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.ArrayList;
import java.util.List;

/**
 * A tunnel is a zone of the board like this:
 *
 * <pre>
 *     $$$$$$
 *          $$$$$
 *     $$$$
 *        $$$$$$$
 * </pre>
 */
public class Tunnel {

    // STATIC

    protected TileInfo start;
    protected TileInfo end;

    // the tile outside the tunnel adjacent to start
    protected TileInfo startOut;

    // the tile outside the tunnel adjacent to end
    protected TileInfo endOut;
    protected List<Room> rooms;

    // true if the tunnel can only be taken by the player
    protected boolean playerOnlyTunnel;
    protected boolean isOneway;


    // DYNAMIC
    protected boolean crateInside = false;



    public void createTunnelExits() {
        if (this.startOut != null) {
            Direction initDir = start.direction(startOut);
            create(start, initDir, startOut);
        }

        if (endOut != null) {
            Direction endDir = end.direction(endOut);
            create(end, endDir, endOut);
        }
    }

    private void create(TileInfo tile, Direction startDir, TileInfo startOut) {
        TileInfo t = tile;

        Direction nextDir = startDir.negate();
        while (true) {
            TileInfo next = t.adjacent(nextDir);

            if (next.isWall() || t.getTunnel() != this) {
                break;
            }

            setExit(t, startDir, startOut);

            t = next;
        }
    }

    private void setExit(TileInfo tile, Direction dir, TileInfo out) {
        if (dir != null) {
            Exit exit = tile.getTunnelExit();

            if (exit == null) {
                exit = new Exit();
                tile.setTunnelExit(exit);
            }

            switch (dir) {
                case RIGHT -> exit.setRightExit(out);
                case UP -> exit.setUpExit(out);
                case DOWN -> exit.setDownExit(out);
                case LEFT -> exit.setLeftExit(out);
            }
        }
    }

    public void addRoom(Room room) {
        if (rooms == null) {
            rooms = new ArrayList<>();
        }
        rooms.add(room);
    }

    public List<Room> getRooms() {
        return rooms;
    }

    public TileInfo getStart() {
        return start;
    }

    public void setStart(TileInfo start) {
        this.start = start;
    }

    public TileInfo getEnd() {
        return end;
    }

    public void setEnd(TileInfo end) {
        this.end = end;
    }

    public TileInfo getStartOut() {
        return startOut;
    }

    public void setStartOut(TileInfo startOut) {
        this.startOut = startOut;
    }

    public TileInfo getEndOut() {
        return endOut;
    }

    public void setEndOut(TileInfo endOut) {
        this.endOut = endOut;
    }

    public boolean isPlayerOnlyTunnel() {
        return playerOnlyTunnel;
    }

    public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
        this.playerOnlyTunnel = playerOnlyTunnel;
    }

    public boolean crateInside() {
        return crateInside;
    }

    public void setCrateInside(boolean crateInside) {
        this.crateInside = crateInside;
    }

    public boolean isOneway() {
        return isOneway;
    }

    public void setOneway(boolean oneway) {
        isOneway = oneway;
    }

    @Override
    public String toString() {
        if (startOut == null) {
            return
              ↪ "closed - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
                    .formatted(start.getX(), start.getY(),
                            end.getX(), end.getY(),
                            endOut.getX(), endOut.getY(),
                            playerOnlyTunnel, isOneway);
        } else if (endOut == null) {
            return
              ↪ "(%d; %d) - (%d; %d) --> (%d; %d) - closed. only player? %s. one way? %s"
                    .formatted(startOut.getX(), startOut.getY(),
                            start.getX(), start.getY(),
                            end.getX(), end.getY(),
                            playerOnlyTunnel, isOneway);
        } else {
            return
              ↪ "(%d; %d) - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
                    .formatted(startOut.getX(), startOut.getY(),
                            start.getX(), start.getY(),
                            end.getX(), end.getY(),
                            endOut.getX(), endOut.getY(),
                            playerOnlyTunnel, isOneway);
        }
    }

    /**
     * Added to every tile that is inside a tunnel.
     * It contains for each direction where is the exit:
     * if you push a crate inside the tunnel to the left, the
     * method {@link #getExit(Direction)} wile return where you will
     * be after pushing the crate until you aren't outside the tunnel.
     *
     * @implNote This object isn't immutable but is assumed as
     * immutable by
↪   MutableBoard.StaticBoard#linkTunnelsRoomsAndTileInfos(MutableBoard.StaticTile[][])
     */
    public static class Exit {

        private TileInfo leftExit;
        private TileInfo upExit;
        private TileInfo rightExit;
        private TileInfo downExit;

        public Exit() {
        }
```

```java

        public Exit(TileInfo leftExit, TileInfo upExit, TileInfo rightExit, TileInfo
        ↪  downExit) {
            this.leftExit = leftExit;
            this.upExit = upExit;
            this.rightExit = rightExit;
            this.downExit = downExit;
        }

        public TileInfo getExit(Direction dir) {
            return switch (dir) {
                case LEFT -> leftExit;
                case UP -> upExit;
                case RIGHT -> rightExit;
                case DOWN -> downExit;
            };
        }

        public TileInfo getLeftExit() {
            return leftExit;
        }

        private void setLeftExit(TileInfo leftExit) {
            this.leftExit = leftExit;
        }

        public TileInfo getUpExit() {
            return upExit;
        }

        private void setUpExit(TileInfo upExit) {
            this.upExit = upExit;
        }

        public TileInfo getRightExit() {
            return rightExit;
        }

        private void setRightExit(TileInfo rightExit) {
            this.rightExit = rightExit;
        }

        public TileInfo getDownExit() {
            return downExit;
        }

        private void setDownExit(TileInfo downExit) {
            this.downExit = downExit;
        }
    }
}
```

### ImmutableBoard

```java
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.solver.board.mark.MarkSystem;
import fr.valax.sokoshell.solver.board.tiles.ImmutableTileInfo;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.List;
```

```java

/**
 * Immutable implementation of {@link Board}.
 *
 * This class extends {@link GenericBoard}. It internally uses {@link ImmutableTileInfo} to store the board content
 * in {@link GenericBoard#content}. As it is immutable, it implements the setters methods always throws a
 * {@link UnsupportedOperationException} when such a method is called.
 *
 * @see Board
 * @see GenericBoard
 * @see TileInfo
 */
public class ImmutableBoard extends GenericBoard {

    public ImmutableBoard(Tile[][] content, int width, int height) {
        super(width, height);

        this.content = new ImmutableTileInfo[height][width];

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                this.content[y][x] = new ImmutableTileInfo(this, content[y][x], x, y);
            }
        }
    }

    public ImmutableBoard(Board other) {
        super(other.getWidth(), other.getHeight());

        this.content = new ImmutableTileInfo[height][width];

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                this.content[y][x] = new ImmutableTileInfo(other.getAt(x, y));
            }
        }
    }

    // GETTERS //

    @Override
    public int getTargetCount() {
        return 0;
    }

    @Override
    public List<Tunnel> getTunnels() {
        return null;
    }

    @Override
    public List<Room> getRooms() {
        return null;
    }

    @Override
    public boolean isGoalRoomLevel() {
        return false;
    }

    @Override
    public MarkSystem getMarkSystem() {
        return null;
    }

    @Override
    public MarkSystem getReachableMarkSystem() {
        return null;
    }
}
```

### Move

```
package fr.valax.sokoshell.solver.board;

/**
 * An enumeration representing a move or a push in a solution. The {@code moveCrate} flag is needed to go back
 * in {@link fr.valax.sokoshell.commands.level.SolutionCommand}
 *
 * DO NOT MODIFY ORDER OF VALUES WITHOUT REMAKING ALL SAVES
 */
public enum Move {

    LEFT("l", Direction.LEFT, false),
    UP("u", Direction.UP, false),
    DOWN("d", Direction.DOWN, false),
    RIGHT("r", Direction.RIGHT, false),

    LEFT_PUSH("L", Direction.LEFT, true),
    UP_PUSH("U", Direction.UP, true),
    RIGHT_PUSH("R", Direction.RIGHT, true),
    DOWN_PUSH("D", Direction.DOWN, true);

    private final String shortName;
    private final Direction direction;
    private final boolean moveCrate;

    Move(String name, Direction direction, boolean moveCrate) {
        this.shortName = name;
        this.direction = direction;
        this.moveCrate = moveCrate;
    }

    public String shortName() {
        return shortName;
    }

    public Direction direction() {
        return direction;
    }

    public boolean moveCrate() {
        return moveCrate;
    }

    public static Move of(Direction dir, boolean moveCrate) {
        return switch (dir) {
            case LEFT -> moveCrate ? LEFT_PUSH : LEFT;
            case UP -> moveCrate ? UP_PUSH : UP;
            case DOWN -> moveCrate ? DOWN_PUSH : DOWN;
            case RIGHT -> moveCrate ? RIGHT_PUSH : RIGHT;
```

```java
        };
    }

    public static Move of(String shortName) {
        for (Move move : Move.values()) {
            if (move.shortName().equals(shortName)) {
                return move;
            }
        }

        return null;
    }
}
```

### Room

```java
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.ArrayList;
import java.util.List;

public class Room {

    protected boolean goalRoom;

    protected final List<TileInfo> tiles = new ArrayList<>();
    protected final List<TileInfo> targets = new ArrayList<>();

    protected List<Tunnel> tunnels;

    /**
     * Only computed if the level is a goal room level as defined by {@link
 ↪   Board#isGoalRoomLevel()}
     */
    protected List<TileInfo> packingOrder;

    // dynamic
    // the index in packingOrder of the position of the next crate that will be pushed
     ↪   inside the room
    // negative if it is not possible because a crate isn't at the correct position
    // or if the room isn't a goal room
    protected int packingOrderIndex;

    public Room() {
    }

    public void addTile(TileInfo tile) {
        tiles.add(tile);

        if (tile.isTarget()) {
            targets.add(tile);
        }
    }


    public List<TileInfo> getTiles() {
        return tiles;
    }

    public List<TileInfo> getTargets() {
        return targets;
    }


    public void addTunnel(Tunnel tunnel) {
        if (tunnels == null) {
            tunnels = new ArrayList<>();
        }
        tunnels.add(tunnel);
    }

    public List<Tunnel> getTunnels() {
        return tunnels;
    }


    public boolean isGoalRoom() {
        return goalRoom;
    }

    public void setGoalRoom(boolean goalRoom) {
        this.goalRoom = goalRoom;
    }

    public List<TileInfo> getPackingOrder() {
        return packingOrder;
    }

    public void setPackingOrder(List<TileInfo> packingOrder) {
        this.packingOrder = packingOrder;
    }

    public boolean isInPackingOrder(TileInfo tile) {
        return packingOrder != null && packingOrder.contains(tile);
    }

    public int getPackingOrderIndex() {
        return packingOrderIndex;
    }

    public void setPackingOrderIndex(int packingOrderIndex) {
        this.packingOrderIndex = packingOrderIndex;
    }
}
```

### MutableBoard

```
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.SokoShell;
import fr.valax.sokoshell.graphics.Surface;
import fr.valax.sokoshell.solver.Corral;
import fr.valax.sokoshell.solver.CorralDetector;
import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.mark.AbstractMarkSystem;
import fr.valax.sokoshell.solver.board.mark.Mark;
import fr.valax.sokoshell.solver.board.mark.MarkSystem;
import fr.valax.sokoshell.solver.board.tiles.GenericTileInfo;
import fr.valax.sokoshell.solver.board.tiles.MutableTileInfo;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
import fr.valax.sokoshell.solver.pathfinder.CratePlayerAStar;
import fr.valax.sokoshell.solver.pathfinder.PlayerAStar;

import java.util.*;
import java.util.function.Consumer;


/**
 * Mutable implementation of {@link Board}.
 *
 * This class extends {@link GenericBoard} by defining all the setters methods. It
↪  internally uses {@link MutableTileInfo} to store the board content
 * in {@link GenericBoard#content}.
 *
 * @see Board
 * @see GenericBoard
 * @see MutableTileInfo
 */
@SuppressWarnings("ForLoopReplaceableByForEach")
public class MutableBoard extends GenericBoard {

    private final MarkSystem markSystem = newMarkSystem(TileInfo::unmark);
    private final MarkSystem reachableMarkSystem = newMarkSystem((t) ->
    ↪  t.setReachable(false));

    private int targetCount;

    /**
     * Tiles that can be 'target' or 'floor'
     */
    private TileInfo[] floors;

    private final List<Tunnel> tunnels = new ArrayList<>();
    private final List<Room> rooms = new ArrayList<>();


    /**
     * True if all rooms are goal room with only one entrance
     */
    private boolean isGoalRoomLevel;

    private PlayerAStar playerAStar;
    private CrateAStar crateAStar;
    private CratePlayerAStar cratePlayerAStar;

    private final CorralDetector corralDetector;

    private StaticBoard staticBoard;

    /**
     * Creates a SolverBoard with the specified width, height and tiles
     *
     * @param content a rectangular matrix of size width * height. The first index is for
↪  the rows
     *                and the second for the columns
     * @param width board width
     * @param height board height
     */
    public MutableBoard(Tile[][] content, int width, int height) {
        super(width, height);

        this.content = new TileInfo[height][width];
```

```java
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                this.content[y][x] = new MutableTileInfo(this, content[y][x], x, y);
            }
        }

        corralDetector = new CorralDetector(this);
    }

    public MutableBoard(int width, int height) {
        super(width, height);

        this.content = new TileInfo[height][width];
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                this.content[y][x] = new MutableTileInfo(this, Tile.FLOOR, x, y);
            }
        }

        corralDetector = new CorralDetector(this);
    }

    /**
     * Creates a copy of 'other'. It doesn't copy solver information
     *
     * @param other the board to copy
     */
    public MutableBoard(Board other) {
        this(other, false);
    }

    public MutableBoard(Board other, boolean copyStatic) {
        super(other.getWidth(), other.getHeight());

        content = new TileInfo[height][width];
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                content[y][x] = new MutableTileInfo(this, other.getAt(x, y));
            }
        }

        corralDetector = new CorralDetector(this);

        if (copyStatic) {
            copyStaticInformation(other);
        }
    }

    private void copyStaticInformation(Board other) {
        // map room in other board and in this board
        Map<Room, Room> roomMap = new HashMap<>(rooms.size());
        Map<Tunnel, Tunnel> tunnelMap = new HashMap<>(rooms.size());

        // copy tunnels, rooms
        for (Room room : other.getRooms()) {
            Room copy = copyRoom(room);
            roomMap.put(room, copy);
            rooms.add(copy);
        }
        for (Tunnel tunnel : other.getTunnels()) {
            Tunnel copy = copyTunnel(tunnel);
            tunnelMap.put(tunnel, copy);
            tunnels.add(copy);
        }

        // copy tile info
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                TileInfo otherTile = other.getAt(x, y);
                TileInfo tile = content[y][x];
                tile.setDeadTile(otherTile.isDeadTile());

                if (tile.getTargets() != null) {
                    tile.setTargets(Arrays.copyOf(tile.getTargets(),
                     ↪  tile.getTargets().length));
                }
                tile.setNearestTarget(otherTile.getNearestTarget());

                tile.setTunnel(tunnelMap.get(otherTile.getTunnel()));
                tile.setRoom(roomMap.get(otherTile.getRoom()));
                if (otherTile.getTunnelExit() != null) {
                    tile.setTunnelExit(otherTile.getTunnelExit()); // it is immutable !
                }
            }
        }

        // link rooms and tunnels
        for (Tunnel tunnel : other.getTunnels()) {
            Tunnel newTunnel = tunnelMap.get(tunnel);
            for (Room room : other.getRooms()) {
                Room newRoom = roomMap.get(room);
                newTunnel.addRoom(newRoom);
                newRoom.addTunnel(newTunnel);
            }
        }
    }

    private Room copyRoom(Room room) {
        Room newRoom = new Room();
        newRoom.setGoalRoom(room.isGoalRoom());

        for (TileInfo t : room.getTiles()) {
            newRoom.addTile(getAt(t.getIndex()));
        }
        if (room.getPackingOrder() != null) {
            List<TileInfo> packingOrder = new ArrayList<>();
            for (TileInfo t : room.getPackingOrder()) {
                packingOrder.add(getAt(t.getIndex()));
            }
            newRoom.setPackingOrder(packingOrder);
        }

        return newRoom;
    }

    private Tunnel copyTunnel(Tunnel tunnel) {
        Tunnel newTunnel = new Tunnel();

        newTunnel.setStart(getAt(tunnel.getStart().getIndex()));
        newTunnel.setEnd(getAt(tunnel.getEnd().getIndex()));

        if (tunnel.getStartOut() != null) {
            newTunnel.setStartOut(getAt(tunnel.getStartOut().getIndex()));
        }
```

```java
        if (tunnel.getEndOut() != null) {
            newTunnel.setEndOut(getAt(tunnel.getEndOut().getIndex()));
        }
        newTunnel.setPlayerOnlyTunnel(tunnel.isPlayerOnlyTunnel());
        newTunnel.setOneway(tunnel.isOneway());

        return newTunnel;
    }

    /**
     * Apply the consumer on every tile info
     *
     * @param consumer the consumer to apply
     */
    public void forEach(Consumer<TileInfo> consumer) {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                consumer.accept(content[y][x]);
            }
        }
    }

    /**
     * Set at tile at the specified index. The index will be converted to
     * cartesian coordinate with {@link #getX(int)} and {@link  #getY(int)}
     *
     * @param index index in the board
     * @param tile the new tile
     * @throws IndexOutOfBoundsException if the index lead to a position outside the board
     */
    public void setAt(int index, Tile tile) { content[getY(index)][getX(index)].setTile(tile); }

    /**
     * Set at tile at (x, y)
     *
     * @param x x position in the board
     * @param y y position in the board
     * @throws IndexOutOfBoundsException if the position is outside the board
     */
    public void setAt(int x, int y, Tile tile) {
        content[y][x].setTile(tile);
    }

    /**
     * Puts the crates of the given state in the content array.
     *
     * @param state The state with the crates
     */
    public void addStateCrates(State state) {
        int[] cratesIndices = state.cratesIndices();
        for (int j = 0; j < cratesIndices.length; j++) {
            int i = cratesIndices[j];
            TileInfo crate = getAt(i);
            crate.setCrateIndex(j);
            crate.addCrate();
        }
    }

    /**
     * Removes the crates of the given state from the content array.
     *
     * @param state The state with the crates
     */
    public void removeStateCrates(State state) {
        for (int i : state.cratesIndices()) {
            TileInfo crate = getAt(i);
            crate.setCrateIndex(-1);
            crate.removeCrate();
        }
    }

    /**
     * Puts the crates of the given state in the content array.
     * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
     *
     * @param state The state with the crates
     */
    public void safeAddStateCrates(State state) {
        for (int i : state.cratesIndices()) {
            TileInfo info = safeGetAt(i);

            if (info != null) {
                info.addCrate();
            }
        }
    }

    /**
     * Removes the crates of the given state from the content array.
     * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
     *
     * @param state The state with the crates
     */
    public void safeRemoveStateCrates(State state) {
        for (int i : state.cratesIndices()) {
            TileInfo info = safeGetAt(i);

            if (info != null) {
                info.removeCrate();
            }
        }
    }

    // ===========================================
    // *         Methods used by solvers         *
    // * You need to call #initForSolver() first *
    // ===========================================

    /**
     * Initialize the board for solving:
     * <ul>
     *     <li>compute floor tiles: an array containing all non-wall tile</li>
     *     <li>compute {@linkplain #computeDeadTiles() dead tiles}</li>
     *     <li>find {@linkplain #findTunnels() tunnels}</li>
     * </ul>
     * <strong>The board must have no crate inside</strong>
     * @see Tunnel
     */
    public void initForSolver() {
        playerAStar = new PlayerAStar(this);
        crateAStar = new CrateAStar(this);
        cratePlayerAStar = new CratePlayerAStar(this);

        computeFloors();
        computeDeadTiles();
        findTunnels();
        findRooms();
        removeUselessTunnels();
        finishComputingTunnels();
        tryComputePackingOrder();
        computeTileToTargetsDistances();

        // we must compute the static board here
        // this is the unique point where the board
        // information are guaranteed to be true.
        // For example, the freeze deadlock detector
        // places wall on the map but this object
        // has no information about this.
        staticBoard = new StaticBoard();
    }

    /**
     * Creates or recreates the floor array. It is an array containing all tile info
     * that are not a wall
     */
    public void computeFloors() {
        int nFloor = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                TileInfo t = getAt(x, y);

                if (!t.isSolid() || t.isCrate()) {
                    nFloor++;
                }
            }
        }

        this.floors = new TileInfo[nFloor];
        int i = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (!this.content[y][x].isSolid() || this.content[y][x].isCrate()) {
                    this.floors[i] = this.content[y][x];
                    i++;
                }
            }
        }
    }

    /**
     * Apply the consumer on every tile info except walls
     *
     * @param consumer the consumer to apply
     */
    public void forEachNotWall(Consumer<TileInfo> consumer) {
        for (TileInfo floor : floors) {
            consumer.accept(floor);
        }
    }

    public void computeTunnelStatus(State state) {
        for (int i = 0; i < tunnels.size(); i++) {
            tunnels.get(i).setCrateInside(false);
```

```java
        }

        for (int i : state.cratesIndices()) {
            Tunnel t = getAt(i).getTunnel();
            if (t != null) {
                // TODO: do the check but need to check if player is between two crates in a tunnel: see boxxle 53
                /*if (t.crateInside()) { // THIS IS VERY IMPORTANT -> see tunnels
                    throw new IllegalStateException();
                }*/

                t.setCrateInside(true);
            }
        }
    }

    public void computePackingOrderProgress(State state) {
        if (!isGoalRoomLevel) {
            return;
        }

        for (int i = 0; i < rooms.size(); i++) {
            rooms.get(i).setPackingOrderIndex(0);
        }

        for (int i : state.cratesIndices()) {
            TileInfo tile = getAt(i);

            Room r = tile.getRoom();
            if (r != null) {
                if (r.isGoalRoom() && tile.isCrate()) { // crate whereas a goal room must contain crate on target
                    r.setPackingOrderIndex(-1);
                }
            }
        }

        for (int i = 0; i < rooms.size(); i++) {
            Room r = rooms.get(i);

            if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
                List<TileInfo> order = r.getPackingOrder();

                // find the first non crate on target tile
                // if the room is completed, then index is equals to -1
                int index = -1;
                for (int j = 0; j < order.size(); j++) {
                    TileInfo tile = order.get(j);

                    if (!tile.isCrateOnTarget()) {
                        index = j;
                        break;
                    }
                }

                // checks that remaining aren't crate on target
                for (int j = index + 1; j < order.size(); j++) {
                    TileInfo tile = order.get(j);

                    if (tile.isCrateOnTarget()) {
                        index = -1;
                        break;
                    }
                }

                r.setPackingOrderIndex(index);
            } else {
                r.setPackingOrderIndex(-1);
            }
        }
    }

    // ************
    // * ANALYSIS *
    // ************

    // * STATIC *

    /**
     * Detects the dead positions of a level. Dead positions are cases that make the level unsolvable
     * when a crate is put on them.
     * After this function has been called, to check if a given crate at (x,y) is a dead position,
     * you can use {@link TileInfo#isDeadTile()} to check in constant time.
     * The board <strong>MUST</strong> have <strong>NO CRATES</strong> for this function to work.
     */
    public void computeDeadTiles() {
        // reset
        forEachNotWall(tile -> tile.setDeadTile(true));

        // loop
        forEachNotWall((tile) -> {
            if (!tile.isDeadTile()) {
                return;
            }

            if (tile.anyCrate()) {
                tile.setDeadTile(true);
                return;
            }

            if (!tile.isTarget()) {
                return;
            }

            findNonDeadCases(tile, null);
        });
    }
    /**
     * Discovers all the reachable cases from (x, y) to find dead positions.
     */
    private void findNonDeadCases(TileInfo tile, Direction lastDir) {
        tile.setDeadTile(false);
        for (Direction d : Direction.VALUES) {
            if (d == lastDir) { // do not go backwards
                continue;
            }

            final int nextX = tile.getX() + d.dirX();
            final int nextY = tile.getY() + d.dirY();
            final int nextNextX = nextX + d.dirX();
            final int nextNextY = nextY + d.dirY();

            if (getAt(nextX, nextY).isDeadTile()    // avoids to check already processed
            ↪   cases
                    && isTileEmpty(nextX, nextY)
                    && isTileEmpty(nextNextX, nextNextY)) {
                findNonDeadCases(getAt(nextX, nextY), d.negate());
            }
        }
    }

    /**
     * Find tunnels. A tunnel is something like this:
     * <pre>
     *     $$$$$$
     *          $$$$$
     *     $$$$
     *        $$$$$$$
     * </pre>
     *
     * A tunnel doesn't contain a target
     */
    public void findTunnels() {
        tunnels.clear();

        markSystem.unmarkAll();
        forEachNotWall((t) -> {
            if (t.isInATunnel() || t.isMarked() || t.isTarget()) {
                return;
            }

            Tunnel tunnel = buildTunnel(t);

            if (tunnel != null) {
                tunnels.add(tunnel);
            }
        });
    }

    /**
     * Try to create a tunnel that contains the specified tile.
     *
     * @param init a tile in the tunnel
     * @return a tunnel that contains the tile or {@code null}
     */
    private Tunnel buildTunnel(TileInfo init) {
        Direction pushDir1 = null;
        Direction pushDir2 = null;

        for (Direction dir : Direction.VALUES) {
            TileInfo adj = init.adjacent(dir);

            if (!adj.isSolid()) {
                if (pushDir1 == null) {
                    pushDir1 = dir;
                } else if (pushDir2 == null) {
                    pushDir2 = dir;
                } else {
                    return null; // too many direction
                }
            }
        }

        if (pushDir1 == null) { // all adjacents tiles are wall, ie init is alone, nerver happen see LevelBuilder
            return null;
        } else if (pushDir2 == null) {
            /*
                We are in this case:
                  |$|
                 $| |$
             */

            Tunnel tunnel = new Tunnel();
            tunnel.setStart(init);
            tunnel.setEnd(init);
            tunnel.setEndOut(init.adjacent(pushDir1));
            init.setTunnel(tunnel);

            growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
            return tunnel;
        } else {
            /*
                Either:
                #| |#
                Either:
                 |#|
                #| |
             */
            boolean onlyPlayer = false;

            if (pushDir1.negate() != pushDir2) {
                /*
                    First case:
                      |#|
                     #|i|
                      | |#
                    if init is like this, then this is a tunnel and a crate
                    mustn't be pushed inside.

                    Second case:
                      |#|
                     #|i|
                      | |
                    ie not tunnel
                */
                if (init.adjacent(pushDir1).adjacent(pushDir2).isSolid()) {
                    onlyPlayer = true;
                } else {
                    return null;
                }
            }

            Tunnel tunnel = new Tunnel();
            tunnel.setEnd(init);
            tunnel.setEndOut(init.adjacent(pushDir1));
            tunnel.setPlayerOnlyTunnel(onlyPlayer);
            init.setTunnel(tunnel);

            growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
            tunnel.setStart(tunnel.getEnd());
            tunnel.setStartOut(tunnel.getEndOut());
            growTunnel(tunnel, init.adjacent(pushDir2), pushDir2);

            return tunnel;
```

```java
        }
    }

    /**
     * Try to grow a tunnel by the end ie Tunnel#end and Tunnel#endOut are modified.
     * The tile adjacent to pos according to -dir is assumed to
     * be a part of a tunnel. So we are in the following situations:
     * <pre>
     *             $$$     $$$
     *     $ $       $     $
     *     $@$     $@$     $@$
     * </pre>
     *
     * @param pos position of the player
     * @param dir the move the player did to go to pos
     */
    private void growTunnel(Tunnel t, TileInfo pos, Direction dir) {
        pos.mark();

        Direction leftDir = dir.left();
        Direction rightDir = dir.right();
        TileInfo left = pos.adjacent(leftDir);
        TileInfo right = pos.adjacent(rightDir);
        TileInfo front = pos.adjacent(dir);

        if (!pos.isTarget()) {
            pos.setTunnel(t);
            if (left.isSolid() && right.isSolid() && front.isSolid()) {
                t.setPlayerOnlyTunnel(true);
                t.setEnd(pos);
                t.setEndOut(null);
                return;
            } else if (left.isSolid() && right.isSolid()) {
                if (front.isMarked()) {
                    t.setEnd(pos);
                    t.setEndOut(front);
                } else {
                    growTunnel(t, front, dir);
                }
                return;
            } else if (right.isSolid() && front.isSolid()) {
                t.setPlayerOnlyTunnel(true);
                if (left.isMarked()) {
                    t.setEnd(pos);
                    t.setEndOut(left);
                } else {
                    growTunnel(t, left, leftDir);
                }
                return;
            } else if (left.isSolid() && front.isSolid()) {
                t.setPlayerOnlyTunnel(true);
                if (right.isMarked()) {
                    t.setEnd(pos);
                    t.setEndOut(right);
                } else {
                    growTunnel(t, right, rightDir);
                }
                return;
            }
        }

        pos.setTunnel(null);
        pos.unmark();
        t.setEndOut(pos);
        t.setEnd(pos.adjacent(dir.negate()));
    }

    /**
     * Finds room based on tunnel. Basically all tile that aren't in a tunnel are in room.
     * This means that you need to call {@link #findTunnels()} before!
     * A room that contains a target is a packing room.
     */
    public void findRooms() {
        forEachNotWall((t) -> {
            if (t.isInATunnel() || t.isInARoom()) {
                return;
            }

            Room room = new Room();
            expandRoom(room, t);
            rooms.add(room);
        });
    }

    private void expandRoom(Room room, TileInfo tile) {
        room.addTile(tile);
        tile.setRoom(room);

        if (tile.isTarget()) {
            room.setGoalRoom(true);
        }

        for (Direction dir : Direction.VALUES) {
            TileInfo adj = tile.adjacent(dir);

            if (!adj.isSolid()) {
                if (!adj.isInATunnel() && !adj.isInARoom()) {
                    expandRoom(room, adj);
                } else if (adj.isInATunnel()) {
                    // avoid add two times a tunnel to a room
                    // It occurs when a tunnel has his two entrance
                    // connected to a room
                    if (room.tunnels == null || !room.tunnels.contains(adj.getTunnel())) {
                        room.addTunnel(adj.getTunnel());
                        adj.getTunnel().addRoom(room);
                    }
                }
            }
        }
    }

    /**
     * Due to this, SokHard 49 can't be solved...
     */
    private void removeUselessTunnels() {
        for (int i = 0; i < tunnels.size(); i++) {
            Tunnel t = tunnels.get(i);
            if (t.getStartOut() == null || t.getEndOut() == null) {
                Room room = t.getRooms().get(0); // tunnel is linked to exactly one room
                room.tunnels.remove(t); // detach the tunnel

                if (room.tunnels.size() == 2 && room.tiles.size() == 1 &&
                ↪   !room.isGoalRoom()) {
                    // room is now useless
```

```java
                    // we are in one of the following cases:
                    // ###    # #
                    //     or #
                    // #_#    #_#
                    // _ indicates the tunnel to remove

                    // dir is the direction the player need to take to exit the tunnel
                    Direction dir;
                    if (t.getStartOut() == null) {
                        dir = t.getEnd().direction(t.getEndOut());
                    } else {
                        dir = t.getStart().direction(t.getStartOut());
                    }

                    Tunnel t1 = room.tunnels.get(0);
                    Tunnel t2 = room.tunnels.get(1);
                    TileInfo roomTile = room.getTiles().get(0);

                    merge(t1, t2, room);
                    if (!roomTile.adjacent(dir).isSolid()) {
                        // second case
                        // tunnel became in every case player only
                        t1.setPlayerOnlyTunnel(true);
                    }

                    // remove t2, taking care of i
                    int j = tunnels.indexOf(t2);
                    tunnels.remove(j);
                    if (j < i) {
                        i--;
                    }
                }

                tunnels.remove(i);
                forEachNotWall((tunnel) -> {
                    if (tunnel.getTunnel() == t) {
                        tunnel.setTunnel(null);
                    }
                });
                i--;
            }
        }
    }

    /**
     * Merge two tunnels, t1 will hold the result.
     * For each tunnel, start, end, startOut, endOut, playerOnlyTunnel, rooms are updated.
     * For each tile in t2, tunnel is replaced by t1
     */
    private void merge(Tunnel t1, Tunnel t2, Room room) {
        TileInfo toAdd = room.getTiles().get(0);

        if (t1.getStartOut() == toAdd) {
            if (t2.getStartOut() == toAdd) {
                t1.setStart(t2.getEnd());
                t1.setStartOut(t2.getEndOut());
            } else {
                t1.setStart(t2.getStart());
                t1.setStartOut(t2.getStartOut());
            }
        } else {
            if (t2.getStartOut() == toAdd) {
                t1.setEnd(t2.getEnd());
                t1.setEndOut(t2.getEndOut());
            } else {
                t1.setEnd(t2.getStart());
                t1.setEndOut(t2.getStartOut());
            }
        }

        forEachNotWall((t) -> {
            if (t.getTunnel() == t2) {
                t.setTunnel(t1);
            }
        });

        toAdd.setRoom(null);
        toAdd.setTunnel(t1);
        t1.setPlayerOnlyTunnel(t1.isPlayerOnlyTunnel() || t2.isPlayerOnlyTunnel());
        t1.rooms.remove(room);
        t2.rooms.remove(room);

        for (Room r : t2.rooms) {
            r.tunnels.remove(t2);
            r.tunnels.add(t1);
        }

        t1.rooms.addAll(t2.rooms);
    }

    private void finishComputingTunnels() {
        for (int i = 0; i < tunnels.size(); i++) {
            Tunnel tunnel = tunnels.get(i);

            // compute tunnel exits
            tunnel.createTunnelExits();

            // compute oneway property
            if (tunnel.getStartOut() == null || tunnel.getEndOut() == null) {
                tunnel.setOneway(true);
            } else {
                tunnel.getStart().addCrate();
                corralDetector.findCorral(this, tunnel.getStartOut().getX(),
                ↪  tunnel.getStartOut().getY());
                tunnel.getStart().removeCrate();

                tunnel.setOneway(!tunnel.getEndOut().isReachable());
            }
        }
    }

    /**
     * Compute packing order. No crate should be on the board
     */
    public void tryComputePackingOrder() {
        isGoalRoomLevel = rooms.size() > 1;

        if (!isGoalRoomLevel) {
            return;
        }

        for (int i = 0; i < rooms.size(); i++) {
            Room r = rooms.get(i);
            if (r.isGoalRoom() && r.getTunnels().size() != 1) {
                isGoalRoomLevel = false;
                break;
            }
        }

        if (isGoalRoomLevel) {
            for (Room r : rooms) {
                if (r.isGoalRoom() && !computePackingOrder(r)) {
                    isGoalRoomLevel = false; // failed to compute packing order for a
                    ↪  room...
                    break;
                }
            }
        }
    }

    /**
     * The room must have only one entrance and a packing room
     * @param room a room
     */
    private boolean computePackingOrder(Room room) {
        markSystem.unmarkAll();

        Tunnel tunnel = room.getTunnels().get(0);
        TileInfo entrance;
        TileInfo inRoom;
        if (tunnel.getStartOut() != null && tunnel.getStartOut().getRoom() == room) {
            entrance = tunnel.getStart();
            inRoom = tunnel.getStartOut();
        } else {
            entrance = tunnel.getEnd();
            inRoom = tunnel.getEndOut();
        }

        List<TileInfo> targets = room.getTargets();
        for (TileInfo t : targets) {
            t.addCrate();
        }

        List<TileInfo> packingOrder = new ArrayList<>();


        List<TileInfo> frontier = new ArrayList<>();
        List<TileInfo> newFrontier = new ArrayList<>();
        frontier.add(entrance);

        List<TileInfo> accessibleCrates = new ArrayList<>();
        findAccessibleCrates(frontier, newFrontier, accessibleCrates);

        while (!accessibleCrates.isEmpty()) {
            boolean hasChanged = false;

            for (int i = 0; i < accessibleCrates.size(); i++) {
                TileInfo crate = accessibleCrates.get(i);
                crate.removeCrate();
                inRoom.addCrate();

                if (crateAStar.hasPath(entrance, null, inRoom, crate)) {
                    accessibleCrates.remove(i);
                    i--;
                    crate.unmark();
                    crate.removeCrate();

                    // discover new accessible crates
                    frontier.add(crate);
                    findAccessibleCrates(frontier, newFrontier, accessibleCrates);

                    packingOrder.add(crate);
                    hasChanged = true;
                } else {
                    crate.addCrate();
                }

                inRoom.removeCrate();
            }

            if (!hasChanged) {
                for (TileInfo t : targets) {
                    t.removeCrate();
                }

                return false;
            }
        }


        for (TileInfo t : targets) {
            t.removeCrate();
        }

        Collections.reverse(packingOrder);
        room.setPackingOrder(packingOrder);

        return true;
    }

    /**
     * Find accessible crates using bfs from lastFrontier.
     *
     * @param lastFrontier starting point of the bfs
     * @param newFrontier a non-null list that will contain the next tile info to visit
     * @param out a list that will contain accessible crates
     */
    private void findAccessibleCrates(List<TileInfo> lastFrontier, List<TileInfo>
    ↪   newFrontier, List<TileInfo> out) {
        newFrontier.clear();

        for (int i = 0; i < lastFrontier.size(); i++) {
            TileInfo tile = lastFrontier.get(i);

            if (!tile.isMarked()) {
                tile.mark();
                if (tile.anyCrate()) {
                    out.add(tile);
                } else {
                    for (Direction dir : Direction.VALUES) {
                        TileInfo adj = tile.adjacent(dir);

                        if (!adj.isMarked() && !adj.isWall()) {
                            newFrontier.add(adj);
                        }
                    }
                }
            }
```

```java
        }

        if (!newFrontier.isEmpty()) {
            findAccessibleCrates(newFrontier, lastFrontier, out);
        } else {
            lastFrontier.clear();
        }
    }

    private void computeTileToTargetsDistances() {

        List<Integer> targetIndices = new ArrayList<>();

        targetCount = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (this.content[y][x].isTarget() || this.content[y][x].isCrateOnTarget()) {
                    targetCount++;
                    targetIndices.add(getIndex(x, y));
                }
            }
        }

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {

                final TileInfo t = getAt(x, y);

                int minDistToTarget = Integer.MAX_VALUE;
                int minDistToTargetIndex = -1;

                getAt(x, y).setTargets(new TileInfo.TargetRemoteness[targetIndices.size()]);

                for (int j = 0; j < targetIndices.size(); j++) {

                    final int targetIndex = targetIndices.get(j);
                    final int d = (t.isFloor() || t.isTarget()
                                   ? playerAStar.findPath(t, getAt(targetIndex), null, null).getDist()
                                   : 0);


                    if (d < minDistToTarget) {
                        minDistToTarget = d;
                        minDistToTargetIndex = j;
                    }

                    getAt(x, y).getTargets()[j] = new TileInfo.TargetRemoteness(targetIndex, d);
                }
                Arrays.sort(getAt(x, y).getTargets());
                getAt(x, y).setNearestTarget(new TileInfo.TargetRemoteness(minDistToTargetIndex, minDistToTarget));
            }
        }
    }





    // * DYNAMIC *

    /**
     * Find reachable tiles
     * @param playerPos The indic of the case on which the player currently is.
     */
    public void findReachableCases(int playerPos) {
        findReachableCases(getAt(playerPos));
    }

    public void findReachableCases(TileInfo tile) {
        reachableMarkSystem.unmarkAll();
        findReachableCases_aux(tile);
    }

    private void findReachableCases_aux(TileInfo tile) {
        tile.setReachable(true);
        for (Direction d : Direction.VALUES) {
            TileInfo adjacent = tile.adjacent(d);

            // the second part of the condition avoids to check already processed cases
            if (!adjacent.isSolid() && !adjacent.isReachable()) {
                findReachableCases_aux(adjacent);
            }
        }
    }



    private int topX = 0;
    private int topY = 0;

    /**
     * This method compute the top left reachable position of the player of pushing a crate
     * at (crateToMoveX, crateToMoveY) to (destX, destY). It is used to calculate the position
     * of the player in a {@link State}.
     * This is also an example of use of {@link MarkSystem}
     *
     * @return the top left reachable position after pushing the crate
     * @see MarkSystem
     * @see Mark
     */
    @Override
    public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
        // temporary move the crate
        crate.removeCrate();
        crateDest.addCrate();

        topX = width;
        topY = height;

        markSystem.unmarkAll();
        topLeftReachablePosition_aux(crate);

        // undo
        crate.addCrate();
        crateDest.removeCrate();

        return topY * width + topX;
    }

    private void topLeftReachablePosition_aux(TileInfo tile) {
        if (tile.getY() < topY || (tile.getY() == topY && tile.getX() < topX)) {
            topX = tile.getX();
            topY = tile.getY();
        }

        tile.mark();
        for (Direction d : Direction.VALUES) {
            TileInfo adjacent = tile.adjacent(d);

            if (!adjacent.isSolid() && !adjacent.isMarked()) {
                topLeftReachablePosition_aux(adjacent);
            }
        }
    }


    // ********************
    // * GETTERS / SETTERS *
    // ********************

    public StaticBoard staticBoard() {
        return staticBoard;
    }

    /**
     * Returns the number of target i.e. tiles on which a crate has to be pushed to solve the level on the board
     * @return the number of target i.e. tiles on which a crate has to be pushed to solve the level on the board
     */
    public int getTargetCount() {
        return targetCount;
    }


    /**
     * Returns all tunnels that are in this board
     *
     * @return all tunnels that are in this board
     */
    public List<Tunnel> getTunnels() {
        return tunnels;
    }

    /**
     * Returns all rooms that are in this board
     *
     * @return all rooms that are in this board
     */
    public List<Room> getRooms() {
        return rooms;
    }

    public boolean isGoalRoomLevel() {
        return isGoalRoomLevel;
    }

    public PlayerAStar getPlayerAStar() {
        return playerAStar;
    }

    public CrateAStar getCrateAStar() {
        return crateAStar;
    }

    public CratePlayerAStar getCratePlayerAStar() {
        return cratePlayerAStar;
    }

    @Override
    public Corral getCorral(TileInfo tile) {
        return corralDetector.findCorral(tile);
    }

    @Override
    public CorralDetector getCorralDetector() {
        return corralDetector;
    }

    /**
     * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking twice  a tile
     *
     * @return a mark system
     * @see MarkSystem
     */
    public MarkSystem getMarkSystem() {
        return markSystem;
    }

    /**
     * Returns the {@linkplain MarkSystem mark system} used by the {@link #findReachableCases(int)} algorithm
     *
     * @return the reachable mark system
     * @see MarkSystem
     */
    public MarkSystem getReachableMarkSystem() {
        return reachableMarkSystem;
    }

    /**
     * Creates a {@linkplain MarkSystem mark system} that apply the specified reset
     * consumer to every <strong>non-wall</strong> {@linkplain TileInfo tile info}
     * that are in this {@linkplain  Board board}.
     *
     * @param reset the reset function
     * @return a new MarkSystem
     * @see MarkSystem
     * @see Mark
     */
    private MarkSystem newMarkSystem(Consumer<TileInfo> reset) {
        return new AbstractMarkSystem() {
            @Override
            public void reset() {
                mark = 0;
                forEachNotWall(reset);
            }
        };
    }

```

```
protected class StaticBoard extends GenericBoard {

    private final List<ImmutableTunnel> tunnels;
    private final List<ImmutableRoom> rooms;

    public StaticBoard() {
        super(MutableBoard.this.width, MutableBoard.this.height);

        StaticTile[][] content = new StaticTile[height][width];
        this.content = content;

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                content[y][x] = new StaticTile(this, MutableBoard.this.content[y][x]);
            }
        }

        tunnels = MutableBoard.this.tunnels.stream()
                .map((t) -> new ImmutableTunnel(this, t)).toList();
        rooms = MutableBoard.this.rooms.stream()
                .map((r) -> new ImmutableRoom(this, r)).toList();

        linkTunnelsRoomsAndTileInfos(content);
    }

    private void linkTunnelsRoomsAndTileInfos(StaticTile[][] content) {
        Map<Room, ImmutableRoom> roomMap = new HashMap<>(rooms.size());
        for (int i = 0; i < rooms.size(); i++) {
            roomMap.put(MutableBoard.this.rooms.get(i), rooms.get(i));
        }

        Map<Tunnel, ImmutableTunnel> tunnelMap = new HashMap<>(tunnels.size());
        for (int i = 0; i < tunnels.size(); i++) {
            tunnelMap.put(MutableBoard.this.tunnels.get(i), tunnels.get(i));
        }

        // add rooms to tunnels
        List<Tunnel> originalTunnel = MutableBoard.this.tunnels;
        for (int i = 0; i < tunnels.size(); i++) {
            ImmutableTunnel t = tunnels.get(i);
            if (originalTunnel.get(i).rooms != null) {
                t.rooms = originalTunnel.get(i).rooms.stream()
                        .map(r -> (Room) roomMap.get(r)).toList();
            }
        }

        // add tunnels to rooms
        List<Room> originalRooms = MutableBoard.this.rooms;
        for (int i = 0; i < rooms.size(); i++) {
            ImmutableRoom r = rooms.get(i);
            if (originalRooms.get(i).tunnels != null) {
                r.tunnels = originalRooms.get(i).tunnels.stream()
                        .map(t -> (Tunnel) tunnelMap.get(t)).toList();
            }
        }

        // add tunnels, rooms to tile info
        for (int y = 0; y < getHeight(); y++) {
            for (int x = 0; x < getWidth(); x++) {
                TileInfo original = MutableBoard.this.content[y][x];
                StaticTile dest = content[y][x];

```

```
                    dest.tunnel = tunnelMap.get(original.getTunnel());
                    dest.room = roomMap.get(original.getRoom());

                    if (original.getTunnelExit() != null) {
                        dest.exit = original.getTunnelExit(); // it is immutable !
                    }
                }
            }
        }

        @Override
        public int getWidth() {
            return MutableBoard.this.getWidth();
        }

        @Override
        public int getHeight() {
            return MutableBoard.this.getHeight();
        }

        @Override
        public int getTargetCount() {
            return MutableBoard.this.getTargetCount();
        }

        @SuppressWarnings("unchecked")
        @Override
        public List<Tunnel> getTunnels() {
            return (List<Tunnel>) ((List<?>) tunnels); // this is black magic
        }

        @SuppressWarnings("unchecked")
        @Override
        public List<Room> getRooms() {
            return (List<Room>) ((List<?>) rooms); // more black magic !
        }

        @Override
        public boolean isGoalRoomLevel() {
            return MutableBoard.this.isGoalRoomLevel();
        }

        @Override
        public MarkSystem getMarkSystem() {
            return null;
        }

        @Override
        public MarkSystem getReachableMarkSystem() {
            return null;
        }
    }

    /**
     * A TileInfo that contains only static information
     */
    protected static class StaticTile extends GenericTileInfo {

        private final boolean deadTile;

        private final TargetRemoteness[] targets;
        private final TargetRemoteness nearestTarget;
```

```java

        private ImmutableTunnel tunnel;
        private ImmutableRoom room;
        private Tunnel.Exit exit;

        public StaticTile(StaticBoard staticBoard, TileInfo tile) {
            super(staticBoard, removeCrate(tile.getTile()), tile.getX(), tile.getY());
            this.deadTile = tile.isDeadTile();

            if (tile.getTargets() == null) {
                targets = null;
            } else {
                targets = Arrays.copyOf(tile.getTargets(), tile.getTargets().length);
            }

            this.nearestTarget = tile.getNearestTarget();
        }

        private static Tile removeCrate(Tile tile) {
            if (tile == Tile.CRATE) {
                return Tile.FLOOR;
            } else if (tile == Tile.CRATE_ON_TARGET) {
                return Tile.TARGET;
            } else {
                return tile;
            }
        }

        @Override
        public boolean isDeadTile() {
            return deadTile;
        }

        @Override
        public boolean isReachable() {
            return false;
        }

        @Override
        public Tunnel getTunnel() {
            return tunnel;
        }

        @Override
        public Tunnel.Exit getTunnelExit() {
            return exit;
        }

        @Override
        public boolean isInATunnel() {
            return tunnel != null;
        }

        @Override
        public Room getRoom() {
            return room;
        }

        @Override
        public boolean isInARoom() {
            return room != null;
        }

        @Override
        public boolean isMarked() {
            return false;
        }

        @Override
        public int getCrateIndex() {
            return -1;
        }

        @Override
        public TargetRemoteness getNearestTarget() {
            return nearestTarget;
        }

        @Override
        public TargetRemoteness[] getTargets() {
            return targets;
        }
    }

    private static class ImmutableTunnel extends Tunnel {

        public ImmutableTunnel(StaticBoard board, Tunnel tunnel) {
            start = board.getAt(tunnel.start.getIndex());
            end = board.getAt(tunnel.end.getIndex());

            if (startOut != null) {
                startOut = board.getAt(tunnel.startOut.getIndex());
            }
            if (endOut != null) {
                endOut = board.getAt(tunnel.endOut.getIndex());
            }
            playerOnlyTunnel = tunnel.isPlayerOnlyTunnel();
            isOneway = tunnel.isOneway();
        }

        @Override
        public void createTunnelExits() {
            throw new UnsupportedOperationException();
        }

        @Override
        public void addRoom(Room room) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setStart(TileInfo start) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setEnd(TileInfo end) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setStartOut(TileInfo startOut) {
            throw new UnsupportedOperationException();
        }
```

```java

        @Override
        public void setEndOut(TileInfo endOut) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setCrateInside(boolean crateInside) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setOneway(boolean oneway) {
            throw new UnsupportedOperationException();
        }

        @Override
        public boolean crateInside() {
            return false;
        }
    }

    private static class ImmutableRoom extends Room {

        public ImmutableRoom(StaticBoard board, Room room) {
            goalRoom = room.isGoalRoom();

            for (TileInfo t : room.getTiles()) {
                tiles.add(board.getAt(t.getIndex()));
            }
            for (TileInfo t : room.getTargets()) {
                targets.add(board.getAt(t.getIndex()));
            }
            if (room.getPackingOrder() != null) {
                packingOrder = new ArrayList<>();
                for (TileInfo t : room.getPackingOrder()) {
                    packingOrder.add(board.getAt(t.getIndex()));
                }
            }
        }

        @Override
        public void addTunnel(Tunnel tunnel) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void addTile(TileInfo tile) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setGoalRoom(boolean goalRoom) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setPackingOrder(List<TileInfo> packingOrder) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void setPackingOrderIndex(int packingOrderIndex) {
            throw new UnsupportedOperationException();
        }

        @Override
        public int getPackingOrderIndex() {
            return -1;
        }
    }
}
```

### Direction

```java
package fr.valax.sokoshell.solver.board;

/**
 * A small but super useful enumeration. Contains all direction: {@link Direction#LEFT}, {@link Direction#UP},
 * {@link Direction#RIGHT} and {@link Direction#DOWN}.
 *
 * @author PoulpogGaz
 * @author darth-mole
 */
public enum Direction {

    LEFT(-1, 0),
    UP(0, -1),
    RIGHT(1, 0),
    DOWN(0, 1);

    /**
     * Directions along the horizontal axis
     */
    public static final Direction[] HORIZONTAL = new Direction[] {LEFT, RIGHT};

    /**
     * Directions along the vertical axis
     */
    public static final Direction[] VERTICAL = new Direction[] {UP, DOWN};

    public static final Direction[] VALUES = new Direction[] {LEFT, UP, RIGHT, DOWN};

    private final int dirX;
    private final int dirY;

    Direction(int dirX, int dirY) {
        this.dirX = dirX;
        this.dirY = dirY;
    }

    public int dirX() { return dirX; }
    public int dirY() { return dirY; }

    /**
     * Rotate the rotation by 90°. For {@link Direction#UP} it returns {@link Direction#LEFT}
     *
     * @return the direction rotated by 90°
     */
    public Direction left() {
        return switch (this) {
            case DOWN -> RIGHT;
            case LEFT -> DOWN;
            case UP -> LEFT;
            case RIGHT -> UP;
        };
    }

    /**
     * Rotate the rotation by -90°. For {@link Direction#UP} it returns {@link Direction#RIGHT}
     *
     * @return the direction rotated by -90°
     */
    public Direction right() {
        return switch (this) {
            case DOWN -> LEFT;
            case LEFT -> UP;
            case UP -> RIGHT;
            case RIGHT -> DOWN;
        };
    }

    /**
     * @return The opposite direction (e.g for {@link Direction#LEFT} it returns {@link Direction#LEFT} etc.)
     */
    public Direction negate() {
        return switch (this) {
            case DOWN -> UP;
            case UP -> DOWN;
            case LEFT -> RIGHT;
            case RIGHT -> LEFT;
        };
    }

    /**
     * Creates a direction from two coordinates.
     * @param dirX If negative, returns {@link Direction#LEFT}, otherwise returns {@link Direction#RIGHT}
     * @param dirY If negative, return {@link Direction#UP}, otherwise returns {@link Direction#DOWN}
     * @return the direction
     */
    public static Direction of(int dirX, int dirY) {
        if (dirX == 0 && dirY == 0) {
            throw new IllegalArgumentException("(0,0) is not a direction");
        } else if (dirX == 0) {
            if (dirY < 0) {
                return UP;
            } else {
                return DOWN;
            }
        } else if (dirX < 0) {
            return LEFT;
        } else {
            return RIGHT;
        }
    }
```

```
}
```

### GenericBoard

```java
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.solver.Corral;
import fr.valax.sokoshell.solver.CorralDetector;
import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.function.Consumer;

/**
 * A {@code package-private} class meant to be use as a base class for {@link Board} implementations.
 * It defines all read-only methods, as well as a way to store the tiles. It is essentially a 2D-array of
 * {@link TileInfo}, the indices being the y and x coordinates (i.e. {@code content[y][x]} is the tile at (x;y)).
 *
 * @see Board
 * @see TileInfo
 */
public abstract class GenericBoard implements Board {

    protected final int width;

    protected final int height;

    protected TileInfo[][] content;

    public GenericBoard(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @SuppressWarnings("CopyConstructorMissesField")
    public GenericBoard(Board other) {
        this(other.getWidth(), other.getHeight());
    }

    @Override
    public int getWidth() { return width; }

    @Override
    public int getHeight() { return height; }

    @Override
    public int getY(int index) { return index / width; }

    @Override
    public int getX(int index) { return index % width; }

    @Override
    public int getIndex(int x, int y) { return y * width + x; }

    @Override
    public TileInfo getAt(int index) {
        return content[getY(index)][getX(index)];
    }

    @Override
    public TileInfo getAt(int x, int y) {
        return content[y][x];
    }


    // SETTERS: throw UnsupportedOperationException as this object is immutable //
    @Override
    public void forEach(Consumer<TileInfo> consumer) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void setAt(int index, Tile tile) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void setAt(int x, int y, Tile tile) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void addStateCrates(State state) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void removeStateCrates(State state) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void safeAddStateCrates(State state) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    @Override
    public void safeRemoveStateCrates(State state) {
        throw new UnsupportedOperationException("Board is immutable");
    }

    // Solver-used methods: throw UnsupportedOperationException as this object is (for now) not to be used by solvers //

    @Override
    public void initForSolver() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void computeFloors() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void forEachNotWall(Consumer<TileInfo> consumer) {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void computeTunnelStatus(State state) {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void computePackingOrderProgress(State state) {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void computeDeadTiles() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void findTunnels() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void findRooms() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void tryComputePackingOrder() {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public void findReachableCases(int playerPos) {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
        throw new UnsupportedOperationException("Board is not intended for solvers");
    }

    @Override
    public Corral getCorral(TileInfo tile) {
        return null;
    }

    @Override
    public CorralDetector getCorralDetector() {
        return null;
    }
}
```

### Board

```java
package fr.valax.sokoshell.solver.board;

import fr.valax.sokoshell.solver.Corral;
import fr.valax.sokoshell.solver.CorralDetector;
import fr.valax.sokoshell.solver.State;
import fr.valax.sokoshell.solver.board.mark.Mark;
import fr.valax.sokoshell.solver.board.mark.MarkSystem;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.List;
import java.util.function.Consumer;

/**
 * Represents the Sokoban board.<br />
 * This interface defines getters setters for the properties of a Sokoban board, e.g. the width, the height etc.
 * Implementations of this interface are meant to be used with a {@link TileInfo} implementation.
 * This class also defines static and dynamic analysis of the Sokoban board, for instance for solving purposes.
 * Such properties are the following:
 * <ul>
 *     <li>Static</li>
 *     <ul>
 *         <li>Dead positions: cases that make the level unsolvable when a crate is pushed on them</li>
 *     </ul>
 *     <li>Dynamic</li>
 *     <ul>
 *         <li>Reachable cases: cases that the player can reach according to his position</li>
 *     </ul>
 * </ul>
 *
 * @see TileInfo
 */
public interface Board {

    int MINIMUM_WIDTH = 5;
    int MINIMUM_HEIGHT = 5;

    // GETTERS //

    /**
     * Returns the width of the board
     *
     * @return the width of the board
     */
    int getWidth();

    /**
     * Returns the height of the board
     *
     * @return the height of the board
     */
    int getHeight();

    /**
     * Returns the number of target i.e. tiles on which a crate has to be pushed to solve the level on the board
     *
     * @return the number of target i.e. tiles on which a crate has to be pushed to solve the level on the board
     */
    int getTargetCount();

    /**
     * Convert an index to a position on the y-axis
     *
     * @param index the index to convert
     * @return the converted position
     */
    int getY(int index);

    /**
     * Convert an index to a position on the x-axis
     *
     * @param index the index to convert
     * @return the converted position
     */
    int getX(int index);

    /**
     * Convert a (x;y) position to an index
     *
     * @param x Coordinate on x-axis
     * @param y Coordinate on y-axis
     * @return the converted index
     */
    int getIndex(int x, int y);

    /**
     * Returns the {@link TileInfo} at the specific index
     *
     * @param index the index of the {@link TileInfo}
     * @return the TileInfo at the specific index
     * @throws IndexOutOfBoundsException if the index lead to a position outside the board
     * @see #getX(int)
     * @see #getY(int)
     * @see #safeGetAt(int)
     */
    TileInfo getAt(int index);

    /**
     * Returns the {@link TileInfo} at the specific index
     *
     * @param index the index of the {@link TileInfo}
     * @return the TileInfo at the specific index or {@code null}
     * if the index represent a position outside the board
     * @see #getX(int)
     * @see #getY(int)
     */
    default TileInfo safeGetAt(int index) {
        int x = getX(index);
        int y = getY(index);

        if (caseExists(x, y)) {
            return getAt(x, y);
        } else {
            return null;
        }
    }

    /**
     * Returns the {@link TileInfo} at the specific position
     *
     * @param x x the of the tile
     * @param y y the of the tile
     * @return the TileInfo at the specific coordinate
     * @throws IndexOutOfBoundsException if the position is outside the board
     * @see #safeGetAt(int, int)
     */
    TileInfo getAt(int x, int y);

    /**
     * Returns the {@link TileInfo} at the specific position
     *
     * @param x x the of the tile
     * @param y y the of the tile
     * @return the TileInfo at the specific index or {@code null}
     * if the index represent a position outside the board
     * @see #getX(int)
     * @see #getY(int)
     */
    default TileInfo safeGetAt(int x, int y) {
        if (caseExists(x, y)) {
            return getAt(x, y);
        } else {
            return null;
        }
    }

    /**
     * Tells whether the case at (x,y) exists or not (i.e. if the case is in the board)
     *
     * @param x x-coordinate
     * @param y y-coordinate
     * @return {@code true} if the case exists, {@code false} otherwise
     */
    default boolean caseExists(int x, int y) {
        return (0 <= x && x < getWidth()) && (0 <= y && y < getHeight());
    }

    /**
     * Same than caseExists(x, y) but with an index
     *
     * @param index index of the case
     * @return {@code true} if the case exists, {@code false} otherwise
     * @see #caseExists(int, int)
     */
    default boolean caseExists(int index) {
        return caseExists(getX(index), getY(index));
    }

    /**
     * Tells whether the tile at the given coordinates is empty or not.
     *
     * @param x x coordinate of the case
     * @param y y coordinate of the case
     * @return {@code true} if empty, {@code false} otherwise
     */
    default boolean isTileEmpty(int x, int y) {
        TileInfo t = getAt(x, y);
        return !t.isSolid();
    }

    /**
     * Checks if the board is solved (i.e. all the crates are on a target).<br />
     * <strong>The crates MUSTileInfo have been put on the board for this function to work as expected.</strong>
     *
     * @return {@code true} if the board is completed, false otherwise
     */
    default boolean isCompletedWith(State s) {
        for (int i : s.cratesIndices()) {
            if (!getAt(i).isCrateOnTarget()) {
                return false;
            }
        }
        return true;
    }

    /**
     * Checks if the board is completed (i.e. all the crates are on a target)
     *
     * @return true if completed, false otherwise
     */
    default boolean isCompleted() {
        for (int y = 0; y < getHeight(); y++) {
            for (int x = 0; x < getWidth(); x++) {
                if (getAt(x, y).isCrate()) {
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Returns all tunnels that are in this board
     *
     * @return all tunnels that are in this board
     */
    List<Tunnel> getTunnels();

    /**
     * Returns all rooms that are in this board
     *
     * @return all rooms that are in this board
     */
    List<Room> getRooms();

    boolean isGoalRoomLevel();

    /**
     * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking twice  a tile
     *
     * @return a mark system
     * @see MarkSystem
     */
    MarkSystem getMarkSystem();

    /**
     * Returns the {@linkplain MarkSystem mark system} used by the {@link #findReachableCases(int)} algorithm
     *
     * @return the reachable mark system
     * @see MarkSystem
     */
    MarkSystem getReachableMarkSystem();


    // SETTERS //


    /**
```

```java
     * Apply the consumer on every tile info
     *
     * @param consumer the consumer to apply
     */
    void forEach(Consumer<TileInfo> consumer);

    /**
     * Set at tile at the specified index. The index will be converted to
     * cartesian coordinate with {@link #getX(int)} and {@link  #getY(int)}
     *
     * @param index index in the board
     * @param tile  the new tile
     * @throws IndexOutOfBoundsException if the index lead to a position outside the board
     */
    void setAt(int index, Tile tile);

    /**
     * Set at tile at (x, y)
     *
     * @param x x position in the board
     * @param y y position in the board
     * @throws IndexOutOfBoundsException if the position is outside the board
     */
    void setAt(int x, int y, Tile tile);

    /**
     * Puts the crates of the given state in the content array.
     *
     * @param state The state with the crates
     */
    void addStateCrates(State state);

    /**
     * Removes the crates of the given state from the content array.
     *
     * @param state The state with the crates
     */
    void removeStateCrates(State state);

    /**
     * Puts the crates of the given state in the content array.
     * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
     *
     * @param state The state with the crates
     */
    void safeAddStateCrates(State state);

    /**
     * Removes the crates of the given state from the content array.
     * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
     *
     * @param state The state with the crates
     */
    void safeRemoveStateCrates(State state);

    // ===========================================
    // *         Methods used by solvers         *
    // * You need to call #initForSolver() first *
    // ===========================================

    /**
     * Initialize the board for solving:
     * <ul>
     *     <li>compute floor tiles: an array containing all non-wall tile</li>
     *     <li>compute {@linkplain #computeDeadTiles() dead tiles}</li>
     *     <li>find {@linkplain #findTunnels() tunnels}</li>
     * </ul>
     * <strong>The board must have no crate inside</strong>
     *
     * @see Tunnel
     */
    void initForSolver();

    /**
     * Creates or recreates the floor array. It is an array containing all tile info
     * that are not a wall
     */
    void computeFloors();

    /**
     * Apply the consumer on every tile info except walls
     *
     * @param consumer the consumer to apply
     */
    void forEachNotWall(Consumer<TileInfo> consumer);

    /**
     * Compute which tunnel contains a crate
     * @param state current state
     */
    void computeTunnelStatus(State state);

    /**
     * Compute packing order progress for each room if the level
     * is a goal room level
     * @param state current state
     */
    void computePackingOrderProgress(State state);

    // ************
    // * ANALYSIS *
    // ************

    // * STATIC *

    /**
     * Detects the dead positions of a level. Dead positions are cases that make the level unsolvable
     * when a crate is put on them.
     * After this function has been called, to check if a given crate at (x,y) is a dead position,
     * you can use {@link TileInfo#isDeadTile()} to check in constant time.
     * The board <strong>MUST</strong> have <strong>NO CRATES</strong> for this function to work.
     */
    void computeDeadTiles();

    /**
     * Find tunnels. A tunnel is something like this:
     * <pre>
     *     $$$$$$
     *          $$$$$
     *     $$$$
     *        $$$$$$$
     * </pre>
     * <p>
     * A tunnel doesn't contain a target
     */
    void findTunnels();

    /**
     * Finds room based on tunnel. Basically all tile that aren't in a tunnel are in room.
     * This means that you need to call {@link #findTunnels()} before!
     * A room that contains a target is a packing room.
     */
    void findRooms();

    /**
     * Compute packing order. No crate should be on the board
     */
    void tryComputePackingOrder();

    // * DYNAMIC *

    /**
     * Find reachable tiles
     *
     * @param playerPos The indic of the case on which the player currently is.
     */
    void findReachableCases(int playerPos);

    /**
     * This method compute the top left reachable position of the player of pushing a crate
     * at crate to crateDest. It is used to calculate the position
     * of the player in a {@link State}.
     * This is also an example of use of {@link MarkSystem}
     *
     * @return the top left reachable position after pushing the crate
     * @see MarkSystem
     * @see Mark
     */
    int topLeftReachablePosition(TileInfo crate, TileInfo crateDest);

    /**
     * @param tile tile
     * @return the corral in which {@code tile} is
     */
    Corral getCorral(TileInfo tile);

    /**
     * @return the {@link CorralDetector} used to find corrals
     */
    CorralDetector getCorralDetector();
}
```

### State

```
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.utils.SizeOf;

import java.util.Arrays;
import java.util.Random;
```

```java

/**
 * A state represents an arrangement of the crates in the board and the location of the player.
 *
 * @implNote <strong>DO NOT MODIFY THE ARRAY AFTER THE INITIALIZATION. THE HASH WON'T BE RECALCULATED</strong>
 * @author darth-mole
 * @author PoulpoGaz
 */
public class State {

    // http://sokobano.de/wiki/index.php?title=Solver#Hash_Function
    // https://en.wikipedia.org/wiki/Zobrist_hashing
    protected static int[][] zobristValues;

    /**
     * @param minSize minSize is the number of tile in the board
     */
    public static void initZobristValues(int minSize) {
        int i;
        if (zobristValues == null) {
            i = 0;
            zobristValues = new int[minSize][2];
        } else if (zobristValues.length < minSize) {
            i = zobristValues.length;
            zobristValues = Arrays.copyOf(zobristValues, minSize);
        } else {
            i = zobristValues.length;
        }

        Random random = new Random();
        for (; i < zobristValues.length; i++) {
            if (zobristValues[i] == null) {
                zobristValues[i] = new int[2];
            }

            zobristValues[i][0] = random.nextInt();
            zobristValues[i][1] = random.nextInt();
        }
    }



    protected final int playerPos;
    protected final int[] cratesIndices;
    protected final int hash;
    protected final State parent;

    public State(int playerPos, int[] cratesIndices, State parent) {
        this(playerPos, cratesIndices, hashCode(playerPos, cratesIndices), parent);
    }

    public State(int playerPos, int[] cratesIndices, int hash, State parent) {
        this.playerPos = playerPos;
        this.cratesIndices = cratesIndices;
        this.hash = hash;
        this.parent = parent;
    }

    /**
     * Creates a child of the state.
     * It uses property of XOR to compute efficiently the hash of the child state
     * @param newPlayerPos the new player position
     * @param crateToMove the index of the crate to move
     * @param crateDestination the new position of the crate to move
     * @return the child state
     */
    public State child(int newPlayerPos, int crateToMove, int crateDestination) {
        int[] newCrates = this.cratesIndices().clone();
        int hash = this.hash ^ zobristValues[this.playerPos][0] ^
         ↪  zobristValues[newPlayerPos][0] // 'moves' the player in the hash
                ^ zobristValues[newCrates[crateToMove]][1] ^
                 ↪  zobristValues[crateDestination][1]; // 'moves' the crate in the hash
        newCrates[crateToMove] = crateDestination;

        return new State(newPlayerPos, newCrates, hash, this);
    }

    public long approxSizeOfAccurate() {
        return SizeOf.getStateLayout().instanceSize() +
                SizeOf.getIntArrayLayout().instanceSize() +
                (long) Integer.BYTES * cratesIndices.length;
    }

    public long approxSizeOf() {
        return 32 +
                16 +
                (long) Integer.BYTES * cratesIndices.length;
    }

    /**
     * The index of the case of the board on which the player is.
     */
    public int playerPos() {
        return playerPos;
    }

    /**
     * The index of the cases of the board on which the crates are.
     */
    public int[] cratesIndices() {
        return cratesIndices;
    }

    public int hash() {
        return hash;
    }

    /**
     * The state in which the board was before coming to this state.
     */
    public State parent() {
        return parent;
    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        State state = (State) o;

```

```java
        if (playerPos != state.playerPos) return false;
        return equals(cratesIndices, state.cratesIndices);
    }

    /**
     * Returns true if all elements of array1 are included in array2 and vice-versa.
     * However, because there is no duplicate and the two array have the same length,
     * it is only necessary to check if array1 is included in array2.
     *
     * @param array1 the first array
     * @param array2 the second array
     * @return true if all elements are included in the second one
     */
    private boolean equals(int[] array1, int[] array2) {
        for (int a : array1) {
            if (!contains(a, array2)) {
                return false;
            }
        }

        return true;
    }

    private boolean contains(int a, int[] array) {
        for (int b : array) {
            if (a == b) {
                return true;
            }
        }

        return false;
    }

    @Override
    public int hashCode() {
        return hash;
    }

    public static int hashCode(int playerPos, int[] cratesIndices) {
        int hash = zobristValues[playerPos][0];

        for (int crate : cratesIndices) {
            hash ^= zobristValues[crate][1];
        }

        return hash;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Player: ").append(playerPos).append(", Crates: [");

        for (int i = 0; i < cratesIndices.length; i++) {
            int crate = cratesIndices[i];
            sb.append(crate);

            if (i + 1 < cratesIndices.length) {
                sb.append("; ");
            }
        }

        sb.append("], hash: ").append(hash);

        return sb.toString();
    }
}
```

### ReachableTiles

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.mark.FixedSizeMarkSystem;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

public class ReachableTiles {

    protected final FixedSizeMarkSystem reachable;

    public ReachableTiles(Board board) {
        reachable = new FixedSizeMarkSystem(board.getWidth() * board.getHeight());
    }

    public boolean isReachable(TileInfo tile) {
        return reachable.isMarked(tile.getIndex());
    }

    public void findReachableCases(TileInfo origin) {
        reachable.unmarkAll();
        findReachableCases_aux(origin);
    }

    private void findReachableCases_aux(TileInfo tile) {
        reachable.mark(tile.getIndex());
        for (Direction d : Direction.VALUES) {
            TileInfo adjacent = tile.adjacent(d);

            // the second part of the condition avoids to check already processed cases
            if (!adjacent.isSolid() && !isReachable(adjacent)) {
                findReachableCases_aux(adjacent);
            }
        }
    }
}
```

### Solver

```java
package fr.valax.sokoshell.solver;

import java.util.List;

/**
 * Defines the basics for all sokoban solver
 *
 * @author darth-mole
 * @author PoulpoGaz
 */
public interface Solver {

    String DFS = "DFS";
    String BFS = "BFS";
    String A_STAR = "A*";

    /**
     * Try to solve the sokoban that is in the {@link SolverParameters}.
     * @param params non null solver parameters
     * @return a solution object
     * @see SolverReport
     * @see SolverParameters
     */
    SolverReport solve(SolverParameters params);

    /**
     * @return the name of solver
     */
    String getName();

    /**
     * @return {@code true} if the solver is running
     */
    boolean isRunning();

    /**
     * Try to stop the solver if it is running.
     * When the solver is not running, it does nothing and returns {@code false}.
     * A solver that doesn't support stopping must return {@code false}
     * @return {@code true} if the solver was stopped, or if it registers the stop action.
     * Otherwise, it returns {@code false}.
     */
    boolean stop();

    /**
     * Returns parameters accepted by this solver.
     * The list returned is always a new one except when the solver don't have any parameter.
     *
     * @return Parameters accepted by this solver.
     */
    List<SolverParameter> getParameters();
}
```

### DeadlockTable

```
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.graphics.style.BasicStyle;
import fr.valax.sokoshell.readers.XSBReader;
import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.MutableBoard;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Function;

public class DeadlockTable {
```

```java

    protected static final int NOT_A_DEADLOCK = 0;
    protected static final int MAYBE_A_DEADLOCK = 1;
    protected static final int A_DEADLOCK = 2;

    protected static final DeadlockTable DEADLOCK = new DeadlockTable(A_DEADLOCK);
    protected static final DeadlockTable NOT_DEADLOCK = new DeadlockTable(NOT_A_DEADLOCK);

    protected final int deadlock;

    protected final int x; // relative to player x
    protected final int y; // relative to player y
    protected final DeadlockTable floorChild;
    protected final DeadlockTable wallChild;
    protected final DeadlockTable crateChild;

    private DeadlockTable(int deadlock) {
        this(deadlock, -1, -1, null, null, null);
    }

    public DeadlockTable(int deadlock, int x, int y,
                         DeadlockTable floorChild, DeadlockTable wallChild, DeadlockTable crateChild) {
        this.deadlock = deadlock;
        this.x = x;
        this.y = y;
        this.floorChild = floorChild;
        this.wallChild = wallChild;
        this.crateChild = crateChild;
    }

    public boolean isDeadlock(TileInfo player, Direction pushDir) {
        Board board = player.getBoard();

        if (player.adjacent(pushDir).isCrateOnTarget()) {
            return false;
        }

        return switch (pushDir) {
            case LEFT -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.y, player.getY() + t.x));
            case UP -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.x, player.getY() + t.y));
            case RIGHT -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.y, player.getY() - t.x));
            case DOWN -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.x, player.getY() - t.y));
        };
    }

    private boolean isDeadlock(Function<DeadlockTable, TileInfo> getTile) {
        if (deadlock == A_DEADLOCK) {
            return true;
        } else if (deadlock == NOT_A_DEADLOCK) {
            return false;
        }

        TileInfo tile = getTile.apply(this);

        if (tile == null) {
            return false;
        }

        return switch (tile.getTile()) {
            case FLOOR -> floorChild.isDeadlock(getTile);
            case WALL -> wallChild.isDeadlock(getTile);
            case CRATE -> crateChild.isDeadlock(getTile);
            default -> false;
        };
    }

    public static void write(DeadlockTable root, Path out) throws IOException {
        try (OutputStream os = new BufferedOutputStream(Files.newOutputStream(out))) {
            Stack<DeadlockTable> stack = new Stack<>();
            stack.push(root);

            while (!stack.isEmpty()) {
                DeadlockTable table = stack.pop();

                os.write(table.deadlock);
                if (table.deadlock == MAYBE_A_DEADLOCK) {
                    writeInt(os, table.x);
                    writeInt(os, table.y);
                    stack.push(table.crateChild);
                    stack.push(table.wallChild);
                    stack.push(table.floorChild);
                }
            }
        }
    }

    public static DeadlockTable read(Path in) throws IOException {
        try (InputStream is = new BufferedInputStream(Files.newInputStream(in))) {
            return read(is);
        }
    }

    private static DeadlockTable read(InputStream is) throws IOException {
        int i = is.read();

        if (i < 0 || i > 2) {
            throw new IOException("Malformed table");
        }

        if (i == A_DEADLOCK) {
            return DEADLOCK;
        } else if (i == NOT_A_DEADLOCK) {
            return NOT_DEADLOCK;
        } else {
            int x = readInt(is);
            int y = readInt(is);

            DeadlockTable floor = read(is);
            DeadlockTable wall = read(is);
            DeadlockTable crate = read(is);

            return new DeadlockTable(MAYBE_A_DEADLOCK, x, y, floor, wall, crate);
        }
    }

    private static void writeInt(OutputStream os, int val) throws IOException {
        os.write(val & 0xFF);
        os.write((val >> 8) & 0xFF);
        os.write((val >> 16) & 0xFF);
        os.write((val >> 24) & 0xFF);
    }

    private static int readInt(InputStream is) throws IOException {
        int a = is.read() & 0xFF;
        int b = is.read() & 0xFF;
        int c = is.read() & 0xFF;
        int d = is.read() & 0xFF;

        return (d << 24) | (c << 16) | (b << 8) | a;
    }

    public static int countNotDetectedDeadlock(DeadlockTable table, int size) {
        Board board = createBoard(size);

        // no dead tiles by default
        board.setAt(1, 1, Tile.TARGET);
        board.setAt(board.getWidth() - 2, board.getHeight() - 2, Tile.TARGET);

        board.computeFloors();
        board.computeDeadTiles();
        board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);

        return countNotDetectedDeadlock(table, board, board.getWidth() / 2, board.getHeight() - 3);
    }

    private static int countNotDetectedDeadlock(DeadlockTable table, Board board, int playerX, int playerY) {
        if (table.deadlock == A_DEADLOCK) {
            State state = createState(board, playerX, playerY);

            // but dead tiles aren't computed...
            if (FreezeDeadlockDetector.checkFreezeDeadlock(board, state)) {
                return 0;
            }

            CorralDetector detector = board.getCorralDetector();
            detector.findCorral(board, playerX, playerY);
            detector.findPICorral(board, state.cratesIndices());

            boolean deadlock = false;
            for (Corral c : detector.getCorrals()) {
                if (c.isDeadlock(state, true)) {
                    deadlock = true;
                    break;
                }
            }

            if (deadlock) {
                return 0;
            } else {
                BasicStyle.XSB_STYLE.print(board, playerX, playerY);

                return 1; // not detected !
            }
        } else if (table.deadlock == MAYBE_A_DEADLOCK) {
            int n = countNotDetectedDeadlock(table.floorChild, board, playerX, playerY);

            board.setAt(playerX + table.x, playerY + table.y, Tile.WALL);
            n += countNotDetectedDeadlock(table.wallChild, board, playerX, playerY);

```

```java
            board.setAt(playerX + table.x, playerY + table.y, Tile.CRATE);
            n += countNotDetectedDeadlock(table.crateChild, board, playerX, playerY);

            board.setAt(playerX + table.x, playerY + table.y, Tile.FLOOR);

            return n;
        } else {
            return 0;
        }
    }


    public static DeadlockTable generate(int size) {
        // if size = 3, returned board looks like:
        // #######
        // #     #
        // #     #
        // #     #
        // #  @  #
        // #######
        // size of generated pattern: size * size
        Board board = createBoard(size);

        board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);

        return generate(board, createOrder(size), 0, board.getWidth() / 2, board.getHeight() - 3);
    }

    public static DeadlockTable generate2(int size, int nThread) {
        Board board = createBoard(size);

        board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);

        ForkJoinPool pool = new ForkJoinPool(nThread <= 0 ? Runtime.getRuntime().availableProcessors() :  nThread);
        GenerateDeadlockTableTask task = new GenerateDeadlockTableTask(board, createOrder(size), 0, board.getWidth() / 2, board.getHeight() - 3, false);

        DeadlockTable table = pool.invoke(task);
        pool.shutdown();

        return table;
    }


    private static DeadlockTable generate(Board board, int[][] order, int index, int playerX, int playerY) {
        // BasicStyle.XSB_STYLE.print(board, playerX, playerY);

        if (isDeadlock_(board, playerX, playerY)) {
            return DEADLOCK;
        } else if (index < order.length) {
            int relativeX = order[index][0];
            int relativeY = order[index][1];

            board.setAt(playerX + relativeX, playerY + relativeY, Tile.WALL);
            DeadlockTable wallChild = generate(board, order, index + 1, playerX, playerY);

            board.setAt(playerX + relativeX, playerY + relativeY, Tile.CRATE);
            DeadlockTable crateChild = generate(board, order, index + 1, playerX, playerY);

            board.setAt(playerX + relativeX, playerY + relativeY, Tile.FLOOR);
            if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
                return NOT_DEADLOCK;
            }

            DeadlockTable floorChild = generate(board, order, index + 1, playerX, playerY);

            return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild, wallChild, crateChild);
        } else {
            return NOT_DEADLOCK;
        }
    }

    private static Board createBoard(int size) {
        Board board = new MutableBoard(size + 4, size + 4);
        State.initZobristValues(board.getWidth() * board.getHeight());

        for (int x = 0; x < board.getWidth(); x++) {
            board.setAt(x, 0, Tile.WALL);
            board.setAt(x, board.getHeight() - 1, Tile.WALL);
        }

        for (int y = 0; y < board.getHeight(); y++) {
            board.setAt(0, y, Tile.WALL);
            board.setAt(board.getWidth() - 1, y, Tile.WALL);
        }

        return board;
    }

    protected static int[][] createOrder(int size) {
        int[][] order = new int[size * size - 2][2];

        boolean odd = size % 2 == 1;
        int i = 0;
        int half = size / 2;
        for (int y = 0; y > -size; y--) {
            for (int x = -half; x < half || (x == half && odd); x++) {
                if (x == 0 && (y == 0 || y == -1)) {
                    continue;
                }

                order[i] = new int[] {x, y};
                i++;
            }
        }


        return order;
    }

    private static class GenerateDeadlockTableTask extends RecursiveTask<DeadlockTable> {

        private static final AtomicInteger COUNTER = new AtomicInteger();
        private static final int total = 4_782_969;

        private final Board board;
        private final int[][] order;
        private final int index;
        private final int playerX;
        private final int playerY;
        private final boolean check;

        public GenerateDeadlockTableTask(Board board, int[][] order, int index, int playerX, int playerY, boolean check) {
            this.board = board;
            this.order = order;
            this.index = index;
            this.playerX = playerX;
            this.playerY = playerY;
            this.check = check;
        }

        @Override
        protected DeadlockTable compute() {
            int n = COUNTER.incrementAndGet();

            if (n % 10_000 == 0) {
                System.out.printf("%.2f%% - %d%n", 100f * n / total, n);
            }

            if (check && isDeadlock_(board, playerX, playerY)) {
                return DEADLOCK;
            } else if (index < order.length) {
                int relativeX = order[index][0];
                int relativeY = order[index][1];

                GenerateDeadlockTableTask wall = subTask(index, Tile.WALL, true);
                GenerateDeadlockTableTask crate = subTask(index, Tile.CRATE, true);

                wall.fork();
                crate.fork();

                DeadlockTable wallChild = wall.join();
                DeadlockTable crateChild = crate.join();

                if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
                    return NOT_DEADLOCK;
                }

                GenerateDeadlockTableTask floor = subTask(index, Tile.FLOOR, false);
                DeadlockTable floorChild = floor.fork().join();

                // the three are never equals to deadlock because
                // it means the current board is a deadlock, and
                // it must be detected by isDeadlock_
                return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild, wallChild, crateChild);

            } else {
                return NOT_DEADLOCK;
            }
        }

        private GenerateDeadlockTableTask subTask(int index, Tile replacement, boolean check) {
            MutableBoard board = new MutableBoard(this.board);
            int relativeX = order[index][0];
            int relativeY = order[index][1];

            board.setAt(playerX + relativeX, playerY + relativeY, replacement);

            return new GenerateDeadlockTableTask(board, order, index + 1, playerX,
            ↪   playerY, check);
        }
    }




    private static boolean isDeadlock_(Board board, int playerX, int playerY) {
        State first = createState(board, playerX, playerY);

        ReachableTiles reachableTiles = new ReachableTiles(board);
        HashSet<State> visited = new HashSet<>();
        Queue<State> toVisit = new ArrayDeque<>();

        visited.add(first);
        toVisit.offer(first);

        boolean deadlock = true;
        while (!toVisit.isEmpty() && deadlock) {
            State parent = toVisit.poll();

            board.addStateCrates(parent);

            if (FreezeDeadlockDetector.checkFreezeDeadlock(board, parent)) {
                board.removeStateCrates(parent);
                continue;
            }

            reachableTiles.findReachableCases(board.getAt(parent.playerPos()));
            deadlock = addChildrenStates(reachableTiles, parent, board, visited, toVisit);
            board.removeStateCrates(parent);
        }

        board.addStateCrates(first);

        return deadlock;
    }

    private static boolean addChildrenStates(ReachableTiles reachableTiles, State parent,
                                             Board board, Set<State> visited, Queue<State>
                                             ↪   toVisit) {
        for (int i = 0; i < parent.cratesIndices().length; i++) {
            TileInfo crate = board.getAt(parent.cratesIndices()[i]);

            for (Direction dir : Direction.VALUES) {
                TileInfo player = crate.adjacent(dir.negate());

                if (!reachableTiles.isReachable(player)) {
                    continue;
                }

                TileInfo dest = crate.adjacent(dir);
                if (dest.isSolid()) {
                    continue;
                }

                State child;
                if (dest.getY() == 1 || dest.getX() == 1 || dest.getX() ==
                ↪   board.getWidth() - 2) {
                    // remove the crate, it is outside the pattern
                    if (parent.cratesIndices().length == 1) {
                        return false; // all crates were moved outside the pattern. not a
                         ↪  deadlock...
                    }

                    int topLeft = board.topLeftReachablePosition(crate, board.getAt(0,
                     ↪  0));

                    child = new State(topLeft,
                     ↪  copyRemoveOneElement(parent.cratesIndices(), i), parent);

                } else {
                    int topLeft = board.topLeftReachablePosition(crate, dest);
                    child = parent.child(topLeft, i, dest.getIndex());
                }

                if (visited.add(child)) {
                    toVisit.add(child);
                }
            }
        }

        return true; // not a deadlock
    }

    private static int[] copyRemoveOneElement(int[] array, int indexToRemove) {
        int[] newArray = new int[array.length - 1];

        int offset = 0;
        for (int i = 0; i < array.length; i++) {
            if (indexToRemove == i) {
                offset = 1;
            } else {
                newArray[i - offset] = array[i];
            }
        }

        return newArray;
    }

    private static State createState(Board board, int playerX, int playerY) {
        List<Integer> ints = new ArrayList<>();

        board.forEach(t -> {
            if (t.anyCrate()) {
                ints.add(t.getIndex());
            }
        });

        return new State(playerY * board.getWidth() + playerX, ints.stream().mapToInt(i ->
         ↪  i).toArray(), null);
    }
}
```

### AStarSolver

```
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.IJsonReader;
import fr.poulpogaz.json.IJsonWriter;
import fr.poulpogaz.json.JsonException;
```

```java
import fr.valax.sokoshell.commands.AbstractCommand;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverPriorityQueue;
import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
import fr.valax.sokoshell.solver.heuristic.Heuristic;
import fr.valax.sokoshell.solver.heuristic.SimpleHeuristic;
import org.jline.reader.Candidate;
import org.jline.reader.LineReader;

import java.io.IOException;
import java.util.List;

public class AStarSolver extends AbstractSolver<WeightedState> {

    private Heuristic heuristic;
    private int lowerBound;

    public AStarSolver() {
        super(A_STAR);
    }

    @Override
    protected void init(SolverParameters parameters) {
        String heuristicName = parameters.getArgument("heuristic");

        if (heuristicName.equalsIgnoreCase("simple")) {
            heuristic = new SimpleHeuristic(board);
        } else {
            heuristic = new GreedyHeuristic(board);
        }

        toProcess = new SolverPriorityQueue();
    }

    @Override
    protected void addInitialState(Level level) {
        final State s = level.getInitialState();
        lowerBound = heuristic.compute(s);

        toProcess.addState(new WeightedState(s, 0, lowerBound));
    }

    @Override
    protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
        if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
            return;
        }

        final int i = board.topLeftReachablePosition(crate, crateDest);
        // The new player position is the crate position
        WeightedState s = toProcess.cachedState().child(i, crate.getCrateIndex(), crateDest.getIndex());
        s.setHeuristic(heuristic.compute(s));

        if (processed.add(s)) {
            toProcess.addState(s);
        }
    }

    @Override
    protected void addParameters(List<SolverParameter> parameters) {
        super.addParameters(parameters);
        parameters.add(new HeuristicParameter());
    }

    @Override
    public int lowerBound() {
        return lowerBound;
    }

    protected static class HeuristicParameter extends SolverParameter {

        private String value;

        public HeuristicParameter() {
            super("heuristic", "The heuristic the solver should use");
        }

        @Override
        public void set(String argument) throws AbstractCommand.InvalidArgument {
            if (argument.equalsIgnoreCase("greedy") ||
             ↪  argument.equalsIgnoreCase("simple")) {
                this.value = argument;
            } else {
                throw new AbstractCommand.InvalidArgument("No such heuristic: " +
                 ↪  argument);
            }
        }

        @Override
        public Object get() {
            return value;
        }

        @Override
        public Object getDefaultValue() {
            return "greedy";
        }

        @Override
        public void toJson(IJsonWriter jw) throws JsonException, IOException {
            jw.value(value);
        }

        @Override
        public void fromJson(IJsonReader jr) throws JsonException, IOException {
            value = jr.nextString();
        }

        @Override
        public void complete(LineReader reader, String argument, List<Candidate>
         ↪  candidates) {
            candidates.add(new Candidate("simple"));
            candidates.add(new Candidate("greedy"));
        }
    }
}
```

### Corral

```
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.Tunnel;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.*;

public class Corral {

    public static final int POTENTIAL_PI_CORRAL = 0;
    public static final int IS_A_PI_CORRAL = 1;
    public static final int NOT_A_PI_CORRAL = 2;

    protected final int id;
    protected final Board board;

    protected int topX;
    protected int topY;

    protected final Set<Corral> adjacentCorrals = new HashSet<>();

    /**
     * All crates that are inside the corral and surrounding the corral
     */
    protected final List<TileInfo> barrier = new ArrayList<>();
    protected final List<TileInfo> crates = new ArrayList<>();
    protected boolean containsPlayer;
    protected boolean adjacentToPlayerCorral; // the player corral is adjacent to itself
    protected int isPICorral;
    protected boolean onlyCrateOnTarget; // true if all crates in crates list are crate on
     ↪  target
    protected boolean isValid = false;


    protected final Set<CorralState> visited = new HashSet<>();
    protected final Queue<CorralState> toVisit = new ArrayDeque<>();
    protected final ReachableTiles reachable;
    protected CorralState currentState;
    protected DeadlockTable deadlockTable;

    public Corral(int id, Board board) {
        this.id = id;
        this.board = board;
        this.reachable = new ReachableTiles(board);
    }

    public boolean isDeadlock(State originalState) {
        return isDeadlock(originalState, false);
    }

    public boolean isDeadlock(State originalState, boolean forceContainsAllCrate) {
        if (!isPICorral() ||
                onlyCrateOnTarget ||
                !forceContainsAllCrate && crates.size() ==
                 ↪  originalState.cratesIndices().length) {
            return false;
        }

        addFrozenCrates(originalState);
        if (!forceContainsAllCrate && crates.size() ==
         ↪  originalState.cratesIndices().length) {
            return false;
        }
```

```java

        boolean deadlock = true;
        CorralState firstState = removeOutsideCrate(originalState);

        visited.add(firstState);
        toVisit.add(firstState);

        while (!toVisit.isEmpty() && deadlock) {
            currentState = toVisit.remove();

            board.addStateCrates(currentState);

            if (FreezeDeadlockDetector.checkFreezeDeadlock(board, currentState)) {
                board.removeStateCrates(currentState);
                continue;
            }

            board.computeTunnelStatus(currentState);
            reachable.findReachableCases(board.getAt(currentState.playerPos()));
            deadlock = addChildrenStates();

            board.removeStateCrates(currentState);

            if (visited.size() >= 1000) {
                deadlock = false;
            }
        }

        visited.clear();
        toVisit.clear();

        // re-add crates
        board.addStateCrates(originalState);

        return deadlock;
    }

    private void addFrozenCrates(State state) {
        for (int i : state.cratesIndices) {
            TileInfo crate = board.getAt(i);

            if (crates.contains(crate)) {
                continue;
            }

            if (isFrozen(crate, Direction.LEFT) && isFrozen(crate, Direction.UP)) {
                crates.add(crate);
            }
        }
    }

    /**
     * True if the crate is almost frozen ie right now it can be moved
     * in the axis: it happens when an adjacent tile on the axis is solid.
     * The adjacent tile must be in the corral is it is a crate
     */
    private boolean isFrozen(TileInfo tile, Direction axis) {
        TileInfo left = tile.adjacent(axis);
        TileInfo right = tile.adjacent(axis.negate());

        return left.isWall() ||
                left.anyCrate() && crates.contains(left) ||
                right.isWall() ||
                right.anyCrate() && crates.contains(right);
    }


    /**
     * @return false if not a deadlock
     */
    private boolean addChildrenStates() {
        int[] cratesIndices = currentState.cratesIndices();

        boolean deadlock = true;
        for (int i = 0; i < cratesIndices.length && deadlock; i++) {
            TileInfo crate = board.getAt(cratesIndices[i]);

            if (crate.isInATunnel()) {
                deadlock = addChildrenStatesInTunnel(i, crate);
            } else {
                deadlock = addChildrenStatesDefault(i, crate);
            }
        }

        return deadlock;
    }

    //
    // THE TWO FOLLOWING METHODS ARE COPIED FROM ABSTRACT SOLVER.
    // I hope that one day, I will change that
    //

    protected boolean addChildrenStatesInTunnel(int crateIndex, TileInfo crate) {
        // the crate is in a tunnel. two possibilities: move to tunnel.startOut or
        ↪  tunnel.endOut
        // this part of the code assume that there is no other crate in the tunnel.
        // normally, this is impossible...

        for (Direction pushDir : Direction.VALUES) {
            TileInfo player = crate.adjacent(pushDir.negate());

            if (reachable.isReachable(player)) {
                TileInfo dest = crate.getTunnelExit().getExit(pushDir);

                if (dest != null && !dest.isSolid()) {
                    if (!addState(crateIndex, crate, dest, pushDir)) {
                        return false; // not a deadlock
                    }
                }
            }
        }

        return true;
    }

    protected boolean addChildrenStatesDefault(int crateIndex, TileInfo crate) {
        for (Direction d : Direction.VALUES) {
            TileInfo crateDest = crate.adjacent(d);
            if (crateDest.isSolid()) {
                continue; // The destination case is not empty
            }

            if (crateDest.isDeadTile()) {
                continue; // Useless to push a crate on a dead position
            }

            TileInfo player = crate.adjacent(d.negate());
            if (!reachable.isReachable(player)) {
                // The player cannot reach the case to push the crate
                // also checks if tile is solid: a solid tile is never reachable
                continue;
            }


            // check for tunnel
            Tunnel tunnel = crateDest.getTunnel();

            // the crate will be pushed inside the tunnel
            if (tunnel != null) {
                if (tunnel.crateInside()) { // pushing inside will lead to a corral deadlock
                    continue;
                }

                // ie the crate can't be pushed to the other extremities of the tunnel
                // however, sometimes (boxxle 24) it is useful to push the crate inside
                // the tunnel. That's why the second addState is done (after this if)
                // and only if this tunnel isn't oneway
                if (!tunnel.isPlayerOnlyTunnel()) {
                    TileInfo newDest = null;
                    Direction pushDir = null;

                    if (crate == tunnel.getStartOut()) {
                        if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate()) {
                            newDest = tunnel.getEndOut();
                            pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
                        }
                    } else {
                        if (tunnel.getStartOut() != null && !tunnel.getStartOut().anyCrate()) {
                            newDest = tunnel.getStartOut();
                            pushDir = tunnel.getStart().direction(tunnel.getStartOut());
                        }
                    }

                    if (newDest != null && !newDest.isDeadTile()) {
                        if (!addState(crateIndex, crate, newDest, pushDir)) {
                            return false;
                        }
                    }
                }

                if (tunnel.isOneway()) {
                    continue;
                }
            }

            if (!addState(crateIndex, crate, crateDest, d)) {
                return false;
            }
        }

        return true;
    }

    /**
     * @return false if not a deadlock
     */
    private boolean addState(int crateIndex, TileInfo crate, TileInfo dest, Direction pushDir) {
        // a crate can be moved outside the corral
        if (!isInCorral(dest)) {
            return false;
        }

        if (deadlockTable.isDeadlock(crate.adjacent(pushDir.negate()), pushDir)) {
            return true; // current state is a deadlock, we need to continue the research
        }

        // all crates of the corral can be moved to a target

        int n = 0;
        for (int i : currentState.cratesIndices()) {
            if (i != crate.getIndex() && board.getAt(i).isCrateOnTarget()) {
                n++;
            }
        }

        if (dest.isTarget() && n + 1 == currentState.cratesIndices.length) { // TODO: crate may be on target
            return false;
        }

        // create sub state
        int newPlayerPos = board.topLeftReachablePosition(crate, dest);
        CorralState sub = currentState.child(newPlayerPos, crateIndex, dest.getIndex());

        if (crate.isCrate() && dest.isTarget()) {
            sub.increaseNumberOnTarget();
        } else if (crate.isCrateOnTarget() && dest.isFloor()) {
            sub.decreaseNumberOnTarget();
        }

        if (visited.add(sub)) {
            toVisit.offer(sub);
        }

        return true;
    }

    /**
     * Remove crates that are not part of the corral
     * and create a new state without these crates
     * @param state current state
     * @return a state without crate outside the corral
     */
    private CorralState removeOutsideCrate(State state) {
        int numOnTarget = 0;

        int[] newCrates = new int[crates.size()];
        int[] oldCrates = state.cratesIndices();
        int j = 0;
        for (int i = 0; i < oldCrates.length; i++) {
            TileInfo crate = board.getAt(oldCrates[i]);
            if (isInCorral(oldCrates[i])) {
                if (crate.isCrateOnTarget()) {
                    numOnTarget++;
                }

                newCrates[j] = oldCrates[i];
                j++;
            } else {
                crate.removeCrate();
            }
        }

        CorralState corralState = new CorralState(state.playerPos(), newCrates, null);
        corralState.setNumOnTarget(numOnTarget);
        return corralState;
    }

    private boolean isInCorral(int crate) {
        TileInfo tile = board.getAt(crate);

        return crates.contains(tile);
    }

    private boolean isInCorral(TileInfo tile) {
        Corral c = board.getCorral(tile);

        if (c == null) {
            return isInCorral(tile.getIndex());
        } else {
            return c == this;
        }
    }

    public int getTopX() {
        return topX;
    }

    public int getTopY() {
        return topY;
    }

    public List<TileInfo> getBarrier() {
        return barrier;
    }

    public List<TileInfo> getCrates() {
        return crates;
    }

    public boolean containsPlayer() {
        return containsPlayer;
    }

    public boolean isPICorral() {
        return isPICorral == IS_A_PI_CORRAL;
    }

    public DeadlockTable getDeadlockTable() {
        return deadlockTable;
    }

    public void setDeadlockTable(DeadlockTable deadlockTable) {
        this.deadlockTable = deadlockTable;
    }

    @Override
    public int hashCode() {
        return id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Corral corral)) return false;

        return id == corral.id;
    }

    private static class CorralState extends State {

        private int numOnTarget;

        public CorralState(int playerPos, int[] cratesIndices, State parent) {
            super(playerPos, cratesIndices, parent);
        }

        public CorralState(int playerPos, int[] cratesIndices, int hash, State parent) {
            super(playerPos, cratesIndices, hash, parent);
        }

        private CorralState(State state) {
            super(state.playerPos, state.cratesIndices, state.hash, state.parent);
        }

        @Override
        public CorralState child(int newPlayerPos, int crateToMove, int crateDestination)
        ↪   {
            return new CorralState(super.child(newPlayerPos, crateToMove,
            ↪   crateDestination));
        }

        public void increaseNumberOnTarget() {
            numOnTarget++;
        }

        public void decreaseNumberOnTarget() {
            numOnTarget--;
        }

        public int getNumOnTarget() {
            return numOnTarget;
        }

        public void setNumOnTarget(int numOnTarget) {
            this.numOnTarget = numOnTarget;
        }
    }
}
```

### BruteforceSolver

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverCollection;

import java.util.ArrayDeque;

/**
 * This class serves as a base class for DFS and BFS solvers, as these class are nearly the same -- the only
 * difference being in the order in which they treat the states (LIFO for DFS and FIFO for BFS).
 */
public abstract class BruteforceSolver extends AbstractSolver<State> {

    public BruteforceSolver(String name) {
        super(name);
    }

    public static DFSSolver newDFSSolver() {
        return new DFSSolver();
    }

    public static BFSSolver newBFSSolver() {
        return new BFSSolver();
    }

    @Override
    protected void addInitialState(Level level) {
        toProcess.addState(level.getInitialState());
    }

    @Override
    protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
        if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
            return;
        }

        final int i = board.topLeftReachablePosition(crate, crateDest);
        // The new player position is the crate position
        State s = toProcess.cachedState().child(i, crate.getCrateIndex(), crateDest.getIndex());

        if (processed.add(s)) {
            toProcess.addState(s);
        }
    }

    @Override
    public int lowerBound() {
        return -1;
    }

    /**
     * Base class for DFS and BFS solvers collection (both of them use {@link ArrayDeque}), the only difference being in
     * which side of the queue is used (end => FIFO => DFS, start => LIFO => BFS)
     */
    private static abstract class BasicBruteforceSolverCollection implements SolverCollection<State> {

        protected final ArrayDeque<State> collection = new ArrayDeque<>();

        protected State cachedState;

        @Override
        public void clear() {
            collection.clear();
        }

        @Override
        public boolean isEmpty() {
            return collection.isEmpty();
        }

        @Override
        public int size() {
            return collection.size();
        }

        @Override
        public void addState(State state) {
            collection.offer(state);
        }

        @Override
        public State peekAndCacheState() {
            cachedState = popState();
            return cachedState;
        }

        @Override
        public State cachedState() {
            return cachedState;
        }
    }

    private static class DFSSolver extends BruteforceSolver {

        public DFSSolver() {
            super(DFS);
        }

        @Override
        protected void init(SolverParameters parameters) {
            toProcess = new DFSSolverCollection();
        }

        private static class DFSSolverCollection extends BasicBruteforceSolverCollection {

            @Override
            public State popState() {
                return collection.removeLast();
            }

            @Override
            public State peekState() {
                return collection.peekLast();
            }
        }
    }

    private static class BFSSolver extends BruteforceSolver {

        public BFSSolver() {
            super(BFS);
        }

        @Override
        protected void init(SolverParameters parameters) {
            toProcess = new BFSSolverCollection();
        }

        private static class BFSSolverCollection extends BasicBruteforceSolverCollection {

            @Override
            public State popState() {
                return collection.removeFirst();
            }

            @Override
            public State peekState() {
                return collection.peekFirst();
            }

        }
    }
}
```

### Tracker

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.DefaultTracker;

/**
 * A tracker is an object that watch a {@link Trackable} and gather solver statistics
 * @see DefaultTracker
 * @see Trackable
 */
public interface Tracker {

    /**
     * The name of the parameter
     * @see SolverParameters
     */
    String TRACKER_PARAM = "tracker";

    /**
     * Get data from a {@link Trackable}
     * @param trackable a trackable from which we get data
     * @see Trackable
     */
    void updateStatistics(Trackable trackable);

    /**
     * Clear all previously gathered statistics
     */
    void reset();

    /**
     * Build a {@link ISolverStatistics} object. It uses the Trackable to get the last data.
     * It is called once at the end of research.
     * @param trackable a trackable from which we get data
     * @return solver statistics
     * @see ISolverStatistics
     */
    ISolverStatistics getStatistics(Trackable trackable);
}
```

### AbstractSolver

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.graphics.style.BasicStyle;
import fr.valax.sokoshell.solver.board.*;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverCollection;
import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
import fr.valax.sokoshell.utils.SizeOf;

import java.io.IOException;
import java.nio.file.Path;
import java.util.*;

/**
 * This class is the base for bruteforce-based solvers, i.e. solvers that use an
↪  exhaustive search to try and find a
 * solution.
 * @author darth-mole
 */
public abstract class AbstractSolver<S extends State> implements Trackable, Solver {

    protected static final String TIMEOUT = "timeout";
    protected static final String MAX_RAM = "max-ram";
    protected static final String ACCURATE = "accurate";

    protected final String name;

    protected final DeadlockTable table;

    protected SolverCollection<S> toProcess;
    protected final Set<State> processed = new HashSet<>();

    protected MutableBoard board;

    private boolean running = false;
    private boolean stopped = false;

    // statistics
    private long timeStart = -1;
    private long timeEnd = -1;
    private int nStateProcessed = -1;
    private int queueSize = -1;
    private Tracker tracker;

    public AbstractSolver(String name) {
        this.name = name;

        try {
            table = DeadlockTable.read(Path.of("4x4.table"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public SolverReport solve(SolverParameters params) {
        Objects.requireNonNull(params);

        // init statistics, timeout and stop
        String endStatus = null;

        running = true;
        stopped = false;

        long timeout = params.getArgument(TIMEOUT);
        long maxRam = params.getArgument(MAX_RAM);
        boolean accurate = params.getArgument(ACCURATE);

        if (accurate) {
            SizeOf.initialize();
        }

        timeStart = System.currentTimeMillis();
        timeEnd = -1;
        nStateProcessed = 0;
        queueSize = 0;

        if (tracker != null) {
            tracker.reset();
        }

        // init the research

        Level level = params.getLevel();

        State.initZobristValues(level.getWidth() * level.getHeight());

        final State initialState = level.getInitialState();
        State finalState = null;

        board = new MutableBoard(level);
        board.removeStateCrates(initialState);
        board.initForSolver();
        board.getCorralDetector().setDeadlockTable(table);

        init(params);
        processed.clear();

        addInitialState(level);

        if (level.getPack().name().equals("XSokoban_90") && level.getIndex() == 3) {
            board.getAt(9, 10).setDeadTile(true);
        }

        while (!toProcess.isEmpty() && !stopped) {
            if (hasTimedOut(timeout)) {
                endStatus = SolverReport.TIMEOUT;
                break;
            }

            if (hasRamExceeded(maxRam, accurate)) {
                endStatus = SolverReport.RAM_EXCEED;
                break;
            }

            S state = toProcess.peekAndCacheState();
            board.addStateCrates(state);

            if (board.isCompletedWith(state)) {
                finalState = state;
                break;
            }
```

```java

            int playerX = board.getX(state.playerPos());
            int playerY = board.getY(state.playerPos());

            CorralDetector detector = board.getCorralDetector();
            detector.findCorral(board, playerX, playerY);

            if (checkPICorralDeadlock(state)) {
                board.removeStateCrates(state);
                continue;
            }

            // compute after checking for corral deadlock, as corral deadlock deals with
             ↪  tunnels
            board.computeTunnelStatus(state);
            board.computePackingOrderProgress(state);

            addChildrenStates(board.getAt(playerX, playerY));
            board.removeStateCrates(state);
        }

        // END OF RESEARCH

        timeEnd = System.currentTimeMillis();
        nStateProcessed = processed.size();
        queueSize = toProcess.size();

        // 'free' ram
        processed.clear();
        toProcess.clear();
        board = null;

        running = false;

        System.out.println("END: " + finalState + " - " + endStatus);

        if (endStatus != null) {
            return SolverReport.withoutSolution(params, getStatistics(), endStatus);
        } else if (stopped) {
            return SolverReport.withoutSolution(params, getStatistics(),
             ↪  SolverReport.STOPPED);
        } else if (finalState != null) {
            return SolverReport.withSolution(finalState, params, getStatistics());
        } else {
            return SolverReport.withoutSolution(params, getStatistics(),
             ↪  SolverReport.NO_SOLUTION);
        }
    }

    /**
     * Initialize the solver. This method is called after the initialization of
     * the board
     */
    protected abstract void init(SolverParameters parameters);

    protected abstract void addInitialState(Level level);

    protected boolean checkPICorralDeadlock(State state) {
        CorralDetector detector = board.getCorralDetector();
        detector.findPICorral(board, state.cratesIndices());

        for (Corral corral : detector.getCorrals()) {
            if (corral.isDeadlock(state)) {
                return true;
            }
        }

        return false;
    }

    protected void addChildrenStates(TileInfo player) {
        Corral playerCorral = board.getCorralDetector().findCorral(player);

        List<TileInfo> crates = playerCorral.getCrates();
        for (int i = 0; i < crates.size(); i++) {
            TileInfo crateTile = crates.get(i);

            // check if the crate is already at his destination
            if (board.isGoalRoomLevel() && crateTile.isInARoom()) {
                Room r = crateTile.getRoom();

                if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
                    continue;
                } else {
                    tryGoalCut(crateTile);
                }
            }

            Tunnel tunnel = crateTile.getTunnel();
            if (tunnel != null) {
                addChildrenStatesInTunnel(crateTile);
            } else {
                addChildrenStatesDefault(crateTile);
            }
        }
    }

    protected void tryGoalCut(TileInfo crate) {
        TileInfo player = board.getAt(currentState().playerPos());

        // only works because rooms have one entry
        CrateAStar crateAStar = board.getCrateAStar();
        List<Room> rooms = board.getRooms();
        for (int i = 0; i < rooms.size(); i++) {
            Room r = rooms.get(i);

            Tunnel tunnel = r.getTunnels().get(0);
            TileInfo entrance;
            if (tunnel.getStartOut().getRoom() == r) {
                entrance = tunnel.getStartOut();
            } else {
                entrance = tunnel.getEndOut();
            }

            if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
                if (crateAStar.hasPath(player, null, crate, entrance)) {
                    addStateCheckForGoalMacro(crate, entrance, null);
                }
            }
        }
    }

    protected void addChildrenStatesInTunnel(TileInfo crate) {
        // the crate is in a tunnel. two possibilities: move to tunnel.startOut or
        ↪  tunnel.endOut
        // this part of the code assume that there is no other crate in the tunnel.
        // normally, this is impossible...

        for (Direction pushDir : Direction.VALUES) {
            TileInfo player = crate.adjacent(pushDir.negate());

            if (player.isReachable()) {
                TileInfo dest = crate.getTunnelExit().getExit(pushDir);

                if (dest != null && !dest.isSolid()) {
                    addStateCheckForGoalMacro(crate, dest, pushDir);
                }
            }
        }
    }

    protected void addChildrenStatesDefault(TileInfo crate) {
        for (Direction d : Direction.VALUES) {

            TileInfo crateDest = crate.adjacent(d);
            if (crateDest.isSolid()) {
                continue; // The destination case is not empty
            }

            if (crateDest.isDeadTile()) {
                continue; // Useless to push a crate on a dead position
            }

            TileInfo player = crate.adjacent(d.negate());
            if (!player.isReachable()) {
                // The player cannot reach the case to push the crate
                // also checks if tile is solid: a solid tile is never reachable
                continue;
            }


            // check for tunnel
            Tunnel tunnel = crateDest.getTunnel();

            // the crate will be pushed inside the tunnel
            if (tunnel != null) {
                if (tunnel.crateInside()) { // pushing inside will lead to a corral
                ↪  deadlock
                    continue;
                }

                // ie the crate can't be pushed to the other extremities of the tunnel
                // however, sometimes (boxxle 24) it is useful to push the crate inside
                // the tunnel. That's why the second addState is done (after this if)
                // and only if this tunnel isn't oneway
                if (!tunnel.isPlayerOnlyTunnel()) {
                    TileInfo newDest = null;
                    Direction pushDir = null;

                    if (crate == tunnel.getStartOut()) {
                        if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate())
                        ↪  {
                            newDest = tunnel.getEndOut();
                            pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
                        }
                    } else {
                        if (tunnel.getStartOut() != null &&
                        ↪   !tunnel.getStartOut().anyCrate()) {
                            newDest = tunnel.getStartOut();
                            pushDir = tunnel.getStart().direction(tunnel.getStartOut());
                        }
                    }

                    if (newDest != null && !newDest.isDeadTile()) {
                        addStateCheckForGoalMacro(crate, newDest, pushDir);
                    }
                }

                if (tunnel.isOneway()) {
                    continue;
                }
            }

            addStateCheckForGoalMacro(crate, crateDest, d);
        }
    }

    protected void addStateCheckForGoalMacro(TileInfo crate, TileInfo dest, Direction
    ↪   pushDir) {
        Room room = dest.getRoom();
        if (room != null && board.isGoalRoomLevel() && room.getPackingOrderIndex() >= 0) {
            // goal macro!
            TileInfo newDest = room.getPackingOrder().get(room.getPackingOrderIndex());

            addState(crate, newDest, null);
        } else {
            addState(crate, dest, pushDir);
        }
    }

    /**
     * Check if the move leads to a deadlock.
     * Only for simple deadlock that don't require
     * lots of computation like PI Corral deadlock
     *
     * @param crate crate to move
     * @param crateDest crate destination
     * @param pushDir push dir of the player. If the move is a macro move,
     *                it is the last push done by the player. It can be null
     * @return true if deadlock
     */
    protected boolean checkDeadlockBeforeAdding(TileInfo crate, TileInfo crateDest,
    ↪   Direction pushDir) {
        crate.removeCrate();
        crateDest.addCrate();

        boolean deadlock = FreezeDeadlockDetector.checkFreezeDeadlock(crateDest);

        if (!deadlock && pushDir != null) {
            deadlock = table.isDeadlock(crateDest.adjacent(pushDir.negate()), pushDir);
        }

        crate.addCrate();
        crateDest.removeCrate();

        return deadlock;
    }
```

```java

    /**
     * Add a state to the processed set. If it wasn't already added, it is added to
     * the toProcess queue. The move is unchecked
     *
     * @param crate crate to move
     * @param crateDest crate destination
     * @param pushDir push dir of the player. If the move is a macro move,
     *                it is the last push done by the player. It can be null
     */
    protected abstract void addState(TileInfo crate, TileInfo crateDest, Direction
    ↪  pushDir);

    protected boolean hasTimedOut(long timeout) {
        return timeout > 0 && timeout + timeStart < System.currentTimeMillis();
    }

    protected boolean hasRamExceeded(long maxRam, boolean accurate) {
        if (maxRam > 0) {
            State curr = currentState();

            if (curr != null) {
                long stateSize;
                long ramUsed;
                if (accurate) {
                    stateSize = curr.approxSizeOfAccurate();
                    ramUsed = SizeOf.approxSizeOfAccurate(processed, stateSize);
                } else {
                    stateSize = curr.approxSizeOf();
                    ramUsed = SizeOf.approxSizeOf(processed, stateSize);
                }

                return ramUsed + toProcess.size() * stateSize >= maxRam;
            }
        }

        return false;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public boolean isRunning() {
        return running;
    }

    @Override
    public boolean stop() {
        stopped = true;
        return true;
    }


    @Override
    public List<SolverParameter> getParameters() {
        List<SolverParameter> params = new ArrayList<>();
        addParameters(params);
        return params;
    }

    /**
     * Add your parameters to the list returned by {@link #getParameters()}
     * @param parameters parameters that will be returned by {@link #getParameters()}
     */
    protected void addParameters(List<SolverParameter> parameters) {
        parameters.add(new SolverParameter.Long(TIMEOUT, "Maximal runtime of the solver",
        ↪   -1));
        parameters.add(new SolverParameter.RamParameter(MAX_RAM, -1));
        parameters.add(new SolverParameter.Boolean(ACCURATE,
        ↪   "Use a more accurate method to calculate ram usage", false));
    }

    private ISolverStatistics getStatistics() {
        ISolverStatistics stats;

        if (tracker != null) {
            stats = Objects.requireNonNull(tracker.getStatistics(this));
        } else {
            stats = new ISolverStatistics.Basic(timeStart, timeEnd);
        }

        return stats;
    }

    @Override
    public State currentState() {
        if (toProcess != null && running) {
            return toProcess.cachedState();
        } else {
            return null;
        }
    }

    @Override
    public Board staticBoard() {
        if (board != null && running) {
            return board.staticBoard();
        } else {
            return null;
        }
    }

    @Override
    public int nStateExplored() {
        if (timeStart < 0) {
            return -1;
        } else if (timeEnd < 0) {
            return processed.size();
        } else {
            return nStateProcessed;
        }
    }

    @Override
    public int currentQueueSize() {
        if (timeStart < 0) {
            return -1;
        } else if (timeEnd < 0 && toProcess != null) {
            return toProcess.size();
        } else {
            return queueSize;
```

```java
        }
    }

    @Override
    public long timeStarted() {
        return timeStart;
    }

    @Override
    public long timeEnded() {
        return timeEnd;
    }

    @Override
    public void setTacker(Tracker tracker) {
        this.tracker = tracker;
    }

    @Override
    public Tracker getTracker() {
        return tracker;
    }

}
```

### SolverParameter

```java
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.IJsonReader;
import fr.poulpogaz.json.IJsonWriter;
import fr.poulpogaz.json.JsonException;
import fr.valax.sokoshell.commands.AbstractCommand;
import org.jline.reader.Candidate;
import org.jline.reader.LineReader;

import java.io.IOException;
import java.util.List;
import java.util.Objects;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * A parameter given to a {@link Solver}. A parameter has a name and a description.
 * It is responsible for parsing arguments and give default value. Implementations
 * can also define how to auto complete and must implements {@link #fromJson(IJsonReader)}
 * and {@link #toJson(IJsonWriter)}
 */
public abstract class SolverParameter {

    protected final String name;
    protected final String description;

    public SolverParameter(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public abstract void set(String argument) throws AbstractCommand.InvalidArgument;

    public abstract Object get();

    public Object getOrDefault() {
        Object o = get();

        if (o == null) {
            o = Objects.requireNonNull(getDefaultValue());
        }

        return o;
    }

    public abstract Object getDefaultValue();

    public boolean hasArgument() {
        return get() != null;
    }


    public void complete(LineReader reader, String argument, List<Candidate> candidates) {

    }

    /**
     * @implNote name is already writen
     */
    public abstract void toJson(IJsonWriter jw) throws JsonException, IOException;

    /**
     * @implNote name is already read
     */
    public abstract void fromJson(IJsonReader jr) throws JsonException, IOException;





    public static class Integer extends SolverParameter {

        protected final int defaultValue;
        protected java.lang.Integer value = null;

        public Integer(String name, int defaultValue) {
            this(name, null, defaultValue);
        }

        public Integer(String name, String description, int defaultValue) {
            super(name, description);
            this.defaultValue = defaultValue;
        }

        @Override
        public void set(String argument) throws AbstractCommand.InvalidArgument {
            try {
                value = java.lang.Integer.parseInt(argument);
            } catch (NumberFormatException e) {
                throw new AbstractCommand.InvalidArgument(e);
            }
        }

        @Override
        public Object get() {
            return value;
        }

        @Override
        public Object getDefaultValue() {
            return defaultValue;
        }

        @Override
        public void toJson(IJsonWriter jw) throws JsonException, IOException {
            if (value != null) {
                jw.value(value);
            }
        }

        @Override
        public void fromJson(IJsonReader jr) throws JsonException, IOException {
            value = jr.nextInt();
        }
    }

    public static class Long extends SolverParameter {

        protected final long defaultValue;
        protected java.lang.Long value = null;

        public Long(String name, long defaultValue) {
            this(name, null, defaultValue);
        }

        public Long(String name, String description, long defaultValue) {
            super(name, description);
            this.defaultValue = defaultValue;
        }

        @Override
        public void set(String argument) throws AbstractCommand.InvalidArgument {
            try {
                value = java.lang.Long.parseLong(argument);
            } catch (NumberFormatException e) {
                throw new AbstractCommand.InvalidArgument(e);
            }
        }

        @Override
        public Object get() {
            return value;
        }

        @Override
        public Object getDefaultValue() {
            return defaultValue;
        }

        @Override
        public void toJson(IJsonWriter jw) throws JsonException, IOException {
            if (value != null) {
                jw.value(value);
            }
        }

        @Override
        public void fromJson(IJsonReader jr) throws JsonException, IOException {
            value = jr.nextLong();
        }
    }


    public static class Boolean extends SolverParameter {

        protected final boolean defaultValue;
        protected java.lang.Boolean value = null;

        public Boolean(String name, boolean defaultValue) {
            this(name, null, defaultValue);
        }

        public Boolean(String name, String description, boolean defaultValue) {
            super(name, description);
            this.defaultValue = defaultValue;
        }

        @Override
        public void set(String argument) throws AbstractCommand.InvalidArgument {
            try {
                int v = java.lang.Integer.parseInt(argument);

                value = v != 0;
            } catch (NumberFormatException e) {
                value = java.lang.Boolean.parseBoolean(argument);
            }
        }

        @Override
        public Object get() {
            return value;
        }

        @Override
        public Object getDefaultValue() {
            return defaultValue;
        }

        @Override
        public void toJson(IJsonWriter jw) throws JsonException, IOException {
            if (value != null) {
                jw.value(value);
            }
        }

        @Override
        public void fromJson(IJsonReader jr) throws JsonException, IOException {
            value = jr.nextBoolean();
        }
    }



```

```java
    public static class RamParameter extends Long {

        private static final Pattern PATTERN = Pattern.compile("^(\\d+)\\s*([gmk])?b$", Pattern.CASE_INSENSITIVE);

        public RamParameter(String name, long defaultValue) {
            super(name, "Maximal ram usage of the solver", defaultValue);
        }

        public RamParameter(String name, String description, long defaultValue) {
            super(name, description, defaultValue);
        }

        @Override
        public void set(String argument) throws AbstractCommand.InvalidArgument {
            Matcher matcher = PATTERN.matcher(argument);

            if (matcher.matches() && matcher.groupCount() >= 1 && matcher.groupCount() <= 2) {
                long r = java.lang.Long.parseLong(matcher.group(1));

                if (matcher.groupCount() == 2) {
                    String unit = matcher.group(2).toLowerCase();

                    r = switch (unit) {
                        case "g" -> r * 1024 * 1024 * 1024;
                        case "m" -> r * 1024 * 1024;
                        case "k" -> r * 1024;
                        default -> throw new AbstractCommand.InvalidArgument("Invalid ram argument");
                    };
                }

                value = r;
            } else {
                throw new AbstractCommand.InvalidArgument("Invalid ram argument");
            }
        }
    }
}
```

### Trackable

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;

/**
 * A solver that implements this interface allows
 * other objects to get information about the current
 * research.
 * <br>
 * Methods are by default non-synchronized and <strong>should not</strong>
 * modify the state of the solver.
 * Implementations are free to violate the first term of the contract
 * <strong>(not the second)</strong>, but they must indicate it.
 */
public interface Trackable extends Solver {

    /**
     * @return the number of state explored or -1
     */
    int nStateExplored();

    /**
     * Returns the size of the queue. The queue contains all
     * states that will be processed in the future. It may return
     * {@code -1} when the Solver doesn't have a queue, or it is
     * impossible to get this information .
     * @return the size of the queue or -1
     */
    int currentQueueSize();

    /**
     * @return lower bound from initial state
     */
    int lowerBound();

    /**
     * @return the time in milliseconds at which the solver was started
     */
    long timeStarted();

    /**
     * @return the time in milliseconds at which the solver finished the research or was stopped
     */
    long timeEnded();

    /**
     * @return the state the solver is processing. It may return null
     */
    State currentState();

    /**
     * @return an immutable board that contains all static information.
     * The board has no crate on it
     */
    Board staticBoard();

    /**
     * Set the {@link Tracker} that is tracking this trackable
     * @param tracker the tracker
     */
    void setTacker(Tracker tracker);

    /**
     * @return the tracker that is tracking this trackable
     */
    Tracker getTracker();
}
```

### CorralDetector

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.*;

/**
 * A union find structure to find corral in a map.
 * The objective of this object is to compute corral,
 * barriers and topY, topX position of each corral.
 */
@SuppressWarnings("ForLoopReplaceableByForEach")
public class CorralDetector {

    private final Corral[] corrals;
    private final int[] parent;
    private final int[] rank;

    private final Set<Corral> currentCorrals;

    private int realNumberOfCorral;

    public CorralDetector(Board board) {
        int size = board.getWidth() * board.getHeight();
        parent = new int[size];
        rank = new int[size];
        corrals = new Corral[size];

        for (int i = 0; i < parent.length; i++) {
            parent[i] = i;
            corrals[i] = new Corral(i, board);
        }

        currentCorrals = new HashSet<>(size);
    }

    /**
     * Find corral. Compute topX, topY. Find the corral that
     * contains the player.
     * Other values (isPICorral, crates, barriers) are not
     * valid after a call to this method. Use {@link #findPICorral(Board, int[])}
     * to revalidate them.
     *
     * @param board the board
     * @param playerX player position x
     * @param playerY player position y
     */
    public void findCorral(Board board, int playerX, int playerY) {
        currentCorrals.clear();

        int h = board.getHeight();
        int w = board.getWidth();

        for (int y = 1; y < h - 1; y++) {
            TileInfo left = board.getAt(0, y);

            for (int x = 1; x < w - 1; x++) {
                TileInfo t = board.getAt(x, y);

                if (!t.isSolid()) {
                    TileInfo up = board.getAt(x, y - 1);

                    if (!up.isSolid() && !left.isSolid()) {
                        addToCorral(t, up);
                        mergeTwoCorrals(up, left);
                    } else if (!up.isSolid()) {
                        addToCorral(t, up);
                    } else if (!left.isSolid()) {
                        addToCorral(t, left);
                    } else {
                        newCorral(t);
                    }
                } else {
                    int i = t.getIndex();
                    parent[i] = -1;
                    rank[i] = -1;
                    corrals[i].isValid = false;
                }

                left = t;
            }
        }

        int playerCorral = find(playerY * board.getWidth() + playerX);
        corrals[playerCorral].containsPlayer = true;

        realNumberOfCorral = currentCorrals.size();
    }

    /**
     * Find PI corral
     * @param board the board
     * @param crates crates on the board
     */
    public void findPICorral(Board board, int[] crates) {
        preComputePICorral(board, crates);

        List<Corral> corrals = new ArrayList<>(currentCorrals);

        for (int i = 0; i < corrals.size(); i++) {
            Corral c = corrals.get(i);

            if (!c.containsPlayer()) {
                if (isPICorral(c)) {
                    c.isPICorral = Corral.IS_A_PI_CORRAL;
                    corrals.remove(i);
                    i--;
                }
            } else {
                c.isPICorral = Corral.NOT_A_PI_CORRAL;
                corrals.remove(i);
                i--;
            }
        }

        for (Corral c : corrals) {
            if (c.isValid && c.isPICorral == Corral.POTENTIAL_PI_CORRAL) {
                mergeWithAdjacents(board, c);
            }
        }
    }

    protected boolean isICorral(Corral corral) {
        for (TileInfo crate : corral.barrier) {
            for (Direction dir : Direction.VALUES) {
                TileInfo crateDest = crate.adjacent(dir);
                if (crateDest.isSolid()) {
                    continue;
                }

                TileInfo player = crate.adjacent(dir.negate());
                if (player.isSolid()) {
                    continue;
                }

                Corral corralDest = findCorral(crateDest);
                Corral playerCorral = findCorral(player);

                if (corralDest == playerCorral) {
                    return false;
                }
            }
        }

        return true;
    }

    protected boolean isPICorral(Corral corral) {
        if (!corral.adjacentToPlayerCorral || corral.adjacentCorrals.size() != 1) {
            return false;
        }

        for (TileInfo crate : corral.barrier) {
            for (Direction dir : Direction.VALUES) {
                TileInfo crateDest = crate.adjacent(dir);
                if (crateDest.isSolid()) {
                    continue;
                }

                TileInfo player = crate.adjacent(dir.negate());
                if (player.isWall()) {
                    continue;
                } else if (player.anyCrate()) {
                    /*if (!corral.crates.contains(player) && !corral.barrier.contains(player)) {
                        return false;
                    }*/
                    continue;
                }

                if (crateDest.isDeadTile()) {
                    continue; // only consider valid moves
                }

                Corral corralDest = findCorral(crateDest);
                Corral playerCorral = findCorral(player);

                if (playerCorral.containsPlayer() && playerCorral == corralDest) {
                    return false;
                }
            }
        }

        return true;
    }

    protected void mergeWithAdjacents(Board board, Corral corral) {
        while (corral.adjacentCorrals.size() > 1) {
            Iterator<Corral> iterator = corral.adjacentCorrals.iterator();
            Corral adj = null;

            while (iterator.hasNext()) {
                adj = iterator.next();

```

```java
                if (adj.isPICorral()) {
                    return;
                }

                if (!adj.containsPlayer) {
                    break;
                }
            }

            corral = fullyMergeTwoCorrals(board, corral, adj);
        }

        if (isPICorral(corral)) {
            corral.isPICorral = Corral.IS_A_PI_CORRAL;
        } else {
            corral.isPICorral = Corral.NOT_A_PI_CORRAL;
        }
    }

    private Corral fullyMergeTwoCorrals(Board board, Corral a, Corral b) {
        Corral corral = mergeTwoCorrals(board.getAt(a.getTopX(), a.getTopY()), board.getAt(b.getTopX(), b.getTopY()));

        if (corral == b) {
            b = a; // this way, we can deal with corral (before a) and b, without doing disjonction.
        }

        // Merge properties. It is assumed that a and b doesn't contain the player
        // topX, topY are already updated
        // the set currentCorrals was also updated.
        corral.adjacentToPlayerCorral |= b.adjacentToPlayerCorral;
        corral.onlyCrateOnTarget &= b.onlyCrateOnTarget;

        // update adjacentCorrals
        // Add all adjacents corral of b to corral, but corral is adjacent to b,
        // we must remove it. The remove is done before addAll because the resulting
        // set is likely to be bigger than b one.
        b.adjacentCorrals.remove(corral);
        // also update adjacent of b
        for (Corral bAdj : b.adjacentCorrals) {
            bAdj.adjacentCorrals.remove(b);

            if (bAdj != corral) {
                bAdj.adjacentCorrals.add(corral);
            }
        }
        corral.adjacentCorrals.remove(b);
        corral.adjacentCorrals.addAll(b.adjacentCorrals);

        // update barrier and crates
        for (TileInfo tile : b.crates) {
            if (!corral.crates.contains(tile)) {
                corral.crates.add(tile);
            }
        }

        // merge the two barrier. Some crates aren't in a barrier.
        for (TileInfo tile : b.barrier) {
            if (!corral.barrier.contains(tile)) {
                corral.barrier.add(tile);
            }
        }


        int[] adjacents = new int[4];
        int size;
        for (int i = 0; i < corral.barrier.size(); i++) {
            TileInfo crate = corral.barrier.get(i);
            size = 0;
            for (Direction dir : Direction.VALUES) {
                TileInfo tile = crate.adjacent(dir);
                if (tile.isSolid()) {
                    continue;
                }

                Corral adj = findCorral(tile);

                boolean new_ = true;
                for (int k = 0; k < size; k++) {
                    if (adjacents[k] == adj.id) {
                        new_ = false;
                        break;
                    }
                }

                if (new_) {
                    adjacents[size] = adj.id;
                    size++;
                }
            }

            if (size <= 1) { // not in barrier !
                corral.barrier.remove(i);
                i--;
            }
        }

        return corral;
    }

    /**
     * Compute adjacent corrals of crates, barriers and various property of Corral
     */
    protected void preComputePICorral(Board board, int[] crates) {
        List<Corral> adj = new ArrayList<>();

        for (int crateI : crates) {
            TileInfo crate = board.getAt(crateI);

            adj.clear();

            // find adjacent corrals
            boolean adjacentToPlayerCorral = false;
            for (Direction dir : Direction.VALUES) {
                TileInfo tile = crate.adjacent(dir);
                if (tile.isSolid()) {
                    continue;
                }

                Corral corral = findCorral(tile);
                // maximal size of adj is 4, so I think that using a list rather than a set is faster
                if (!adj.contains(corral)) {
                    adj.add(corral);
                }

                if (corral.containsPlayer()) {
                    adjacentToPlayerCorral = true;
                }
            }

            if (adj.size() == 1) {
                // the crate is inside a corral
                // and not a part of a barrier
                adj.get(0).crates.add(crate);

                if (crate.isCrate()) {
                    adj.get(0).onlyCrateOnTarget = false;
                }
            } else if (adj.size() > 1) {
                // crate is a part of a barrier
                for (int i = 0; i < adj.size(); i++) {
                    Corral corral = adj.get(i);
                    corral.crates.add(crate);
                    corral.barrier.add(crate);
                    corral.adjacentToPlayerCorral |= adjacentToPlayerCorral;

                    if (crate.isCrate()) {
                        corral.onlyCrateOnTarget = false;
                    }

                    for (int j = i + 1; j < adj.size(); j++) {
                        Corral corral2 = adj.get(j);

                        if (corral.adjacentCorrals.add(corral2)) {
                            corral2.adjacentCorrals.add(corral);
                        }
                    }
                }
            }
        }
    }

    /**
     * Move a node from a aPackage to another. {@code node}
     * and {@code dest} must be in separate trees.
     * This method breaks the union find structure.
     * So, it must be used carefully.
     */
    private void addToCorral(TileInfo tile, TileInfo inCorral) {
        int i = tile.getIndex();
        int rootI = find(inCorral.getIndex());

        parent[i] = rootI;
        rank[i] = 0;
        rank[rootI] = Math.max(1, rank[rootI]);
    }

    /**
     * Remove a node from his aPackage and create a new aPackage.
     * This method breaks the union find structure.
     * So, it must be used carefully.
     */
    private void newCorral(TileInfo tile) {
        int i = tile.getIndex();
        parent[i] = i;
        rank[i] = 0;

        Corral corral = corrals[i];
        corral.containsPlayer = false;
        corral.isPICorral = Corral.POTENTIAL_PI_CORRAL;
        corral.onlyCrateOnTarget = true;
        corral.isValid = true;
        corral.crates.clear();
        corral.barrier.clear();
        corral.adjacentCorrals.clear();
        corral.topX = tile.getX();
        corral.topY = tile.getY();

        currentCorrals.add(corral);
    }

    private Corral mergeTwoCorrals(TileInfo inCorral1, TileInfo inCorral2) {
        int corral1I = find(inCorral1.getIndex());
        int corral2I = find(inCorral2.getIndex());

        if (corral1I != corral2I) {
            int oldCorralI;
            int newCorralI;
            if (rank[corral1I] < rank[corral2I]) {
                oldCorralI = corral1I;
                newCorralI = corral2I;
            } else if (rank[corral1I] > rank[corral2I]) {
                oldCorralI = corral2I;
                newCorralI = corral1I;
            } else {
                oldCorralI = corral1I;
                newCorralI = corral2I;
                rank[newCorralI]++;
            }

            parent[oldCorralI] = newCorralI;

            Corral newCorral = corrals[newCorralI];
            Corral oldCorral = corrals[oldCorralI];

            oldCorral.isValid = false;
            currentCorrals.remove(oldCorral);
            newCorral.containsPlayer |= oldCorral.containsPlayer();

            if (oldCorral.topY < newCorral.topY || (oldCorral.topY == newCorral.topY &&
            ↪   oldCorral.topX < newCorral.topX)) {
                newCorral.topX = oldCorral.topX;
                newCorral.topY = oldCorral.topY;
            }

            return newCorral;
        }

        return corrals[corral1I];
    }


    private int find(int i) {
        if (parent[i] != i) {
            int root = find(parent[i]);
            parent[i] = root;
```

```java

            return root;
        }

        return i;
    }

    /**
     * The tile must be a non-solid tile: a floor or a target
     * @param tile a floor or target tile
     * @return the corral in which the tile is
     */
    public Corral findCorral(TileInfo tile) {
        int i = tile.getIndex();

        if (parent[i] < 0) {
            return null;
        }

        return corrals[find(i)];
    }

    public Collection<Corral> getCorrals() {
        return currentCorrals;
    }

    public int getRealNumberOfCorral() {
        return realNumberOfCorral;
    }

    public void setDeadlockTable(DeadlockTable table) {
        for (Corral c : corrals) {
            c.setDeadlockTable(table);
        }
    }
}
```

### WeightedState

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.utils.SizeOf;

/**
 * A simple derivation of State with a weight, i.e. something to rank the states.
 * Used for instance by {@link AStarSolver}
 */
public class WeightedState extends State {

    private int cost = 0;

    private int heuristic = 0;

    public WeightedState(int playerPos, int[] cratesIndices, int hash, State parent, int cost, int heuristic) {
        super(playerPos, cratesIndices, hash, parent);
        this.setCost(cost);
        this.setHeuristic(heuristic);
    }

    public WeightedState(State state, int cost, int heuristic) {
        this(state.playerPos(), state.cratesIndices(), state.hash(), state.parent(), cost,
         ↪  heuristic);
    }

    /**
     * <strong>This function does NOT compute the heuristic of the child state.</strong>
     * Use {@link WeightedState#setHeuristic(int)} to set it after calling this method.
     */
    public WeightedState child(int newPlayerPos, int crateToMove, int crateDestination) {
        return new WeightedState(super.child(newPlayerPos, crateToMove, crateDestination),
                cost(), 0);
    }

    @Override
    public long approxSizeOfAccurate() {
        return SizeOf.getWeightedStateLayout().instanceSize() +
                SizeOf.getIntArrayLayout().instanceSize() +
                (long) Integer.BYTES * cratesIndices.length;
    }

    @Override
    public long approxSizeOf() {
        return 40 +
                16 +
                (long) Integer.BYTES * cratesIndices.length;
    }

    /**
     * The state weight, which is the sum of its cost and its heuristic.
     */
    public int weight() {
        return cost() + heuristic();
    }

    /**
     * The cost the come to this state.
     */
    public int cost() {
        return cost;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }

    /**
     * The heuristic between this state and a solution.
     */
    public int heuristic() {
        return heuristic;
    }

    public void setHeuristic(int heuristic) {
        this.heuristic = heuristic;
    }
}
```

### Level

```java
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.JsonException;
import fr.poulpogaz.json.JsonPrettyWriter;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.ImmutableBoard;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.utils.BuilderException;
import fr.valax.sokoshell.utils.Utils;

import java.io.IOException;
import java.math.BigInteger;
import java.util.*;

/**
 * @author darth-mole
 * @author PoulpoGaz
 */
public class Level extends ImmutableBoard {

    // package private
    Pack pack;
    private final int playerPos;
    private final int index;

    private final List<SolverReport> solverReports;

    // number of crate or crate on target
    private final int numberOfCrates;

    // number of crate, crate on target, floor and target
    private final int numberOfNonWalls;

    private BigInteger maxNumberOfStateEstimation;

    public Level(Tile[][] tiles, int width, int height, int playerPos, int index) {
        super(tiles, width, height);
        this.playerPos = playerPos;
        this.index = index;

        solverReports = new ArrayList<>();

        int numCrate = 0;
        int numFloor = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (getAt(x, y).anyCrate()) {
                    numCrate++;
                }
                if (!getAt(x, y).isWall()) {
                    numFloor++;
                }
            }
        }

        this.numberOfCrates = numCrate;
        this.numberOfNonWalls = numFloor;
    }

    public void writeSolutions(JsonPrettyWriter jpw) throws JsonException, IOException {
        for (SolverReport solution : solverReports) {
            jpw.beginObject();
            solution.writeSolution(jpw);
            jpw.endObject();
        }
    }

    /**
     * Returns the player position on the x-axis at the beginning
     *
     * @return the player position on the x-axis at the beginning
     */
    public int getPlayerX() {
        return playerPos % getWidth();
    }

    /**
     * Returns the player position on the y-axis at the beginning
     *
     * @return the player position on the y-axis at the beginning
     */
    public int getPlayerY() {
        return playerPos / getWidth();
    }

    /**
     * Returns the initial state i.e. a state representing the level at the beginning
     *
     * @return the initial state
     */
    public State getInitialState() {
        State.initZobristValues(getWidth() * getHeight()); // TODO

        List<Integer> cratesIndices = new ArrayList<>();

        for (int y = 0; y < getHeight(); y++) {
            for (int x = 0; x < getWidth(); x++) {
                if (getAt(x, y).anyCrate()) {
                    cratesIndices.add(y * getWidth() + x);
                }
            }
        }

        int[] cratesIndicesArray = new int[cratesIndices.size()];
        for (int i = 0; i < cratesIndices.size(); i++) {
            cratesIndicesArray[i] = cratesIndices.get(i);
        }

        return new State(playerPos, cratesIndicesArray, null);
    }

    public BigInteger estimateNumberOfState() {
        if (maxNumberOfStateEstimation == null) {
            // + 1 for numberOfCrate because we also consider the player
            maxNumberOfStateEstimation = Utils.binomial(numberOfNonWalls, numberOfCrates + 1);
        }

        return maxNumberOfStateEstimation;
    }

    public BigInteger estimateNumberOfState(int nDeadTile) {
        int nFloor = numberOfNonWalls - nDeadTile;

        return Utils.binomial(nFloor, numberOfCrates + 1);
    }

    /**
     * @return the number of crate in this level
     */
    public int getNumberOfCrates() {
        return numberOfCrates;
    }

    /**
     * @return the number of non-wall (floor, target, crate, crate on target)
     */
    public int getNumberOfNonWalls() {
        return numberOfNonWalls;
    }

    /**
     * Returns the last solver report that is a solution
     * @return the last solver report that is a solution
     */
    public SolverReport getLastSolution() {
        if (solverReports.isEmpty()) {
            return null;
        }

        for (int i = solverReports.size() - 1; i >= 0; i--) {
            SolverReport r = solverReports.get(i);

            if (r.isSolved()) {
                return r;
            }
        }

        return null;
    }

    /**
     * Returns the last report
     *
     * @return the last report
     */
    public SolverReport getLastReport() {
        if (solverReports.isEmpty()) {
            return null;
        } else {
            return solverReports.get(solverReports.size() - 1);
        }
    }

    /**
     * Returns the solver report at the specified position
     *
     * @param index index of the report to return
     * @return the solver report at the specified position
     */
    public SolverReport getSolverReport(int index) {
        if (index < 0 || index >= solverReports.size()) {
            return null;
        } else {
            return solverReports.get(index);
        }
    }

    /**
```

```
     * Returns all solver reports
     *
     * @return all solver reports
     */
    public List<SolverReport> getSolverReports() {
        return solverReports;
    }

    /**
     * Returns the number of solver report
     *
     * @return the number of solver report
     */
    public int numberOfSolverReport() {
        return solverReports.size();
    }

    /**
     * Add a solver report to this level
     *
     * @param solverReport the report to add
     * @throws IllegalArgumentException if the report isn't for this level
     */
    public synchronized void addSolverReport(SolverReport solverReport) {
        if (solverReport.getParameters().getLevel() != this) {
            throw new
             ↪  IllegalArgumentException("Attempting to add a report to the wrong level");
        }
        solverReports.add(solverReport);
    }

    public synchronized void removeSolverReport(int index) {
        solverReports.remove(index);
    }

    public synchronized int indexOf(SolverReport solverReport) {
        if (solverReport.getParameters().getLevel() != this) {
            return -1;
        }
        return solverReports.indexOf(solverReport);
    }

    /**
     * Returns if an attempt to solve this level was done. It doesn't mean that this level
↪   has a solution
     *
     * @return {@code true} if an attempt to solve this level was done.
     */
    public boolean hasReport() {
        return solverReports.size() > 0;
    }

    /**
     * Returns {@code true} if this level has a solution
     *
     * @return {@code true} if this level has a solution
     */
    public boolean hasSolution() {
        for (int i = 0; i < solverReports.size(); i++) {
            SolverReport r = solverReports.get(i);
            if (r.isSolved()) {
                return true;
            }
        }

        return false;
    }

    /**
     * Returns the index of this level in the pack
     *
     * @return the index of this level in the pack
     */
    public int getIndex() {
        return index;
    }

    /**
     * Returns the pack in which this level is
     *
     * @return the pack in which this level is
     */
    public Pack getPack() {
        return pack;
    }


    /**
     * A builder of {@link Level}
     */
    public static class Builder {

        private int playerX = -1;
        private int playerY = -1;

        private Tile[][] board = new Tile[0][0];
        private int width;
        private int height;
        private int index;

        /**
         * Builds and returns a {@link Level}
         *
         * @return the new {@link Level}
         * @throws BuilderException if the player is outside the board
         * r the player is on a solid tile
         */
        public Level build() {
            if (board == null) {
                throw new BuilderException("Board is null");
            }

            if (playerX < 0 || playerX >= width) {
                throw new BuilderException("Player x out of bounds");
            }

            if (playerY < 0 || playerY >= height) {
                throw new BuilderException("Player y out of bounds");
            }

            if (board[playerY][playerX].isSolid()) {
                throw new BuilderException("Player is on a solid tile");
            }

```

```java
        formatLevel();

        return new Level(board, width, height, playerY * width + playerX, index);
    }

    /**
     * Format the level for the solver. Some levels aren't surrounded by wall
     * or have rooms that are inaccessible. This method removes these rooms
     * and add wall if necessary.
     */
    private void formatLevel() {
        Set<Integer> visited = new HashSet<>();

        int i = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (board[y][x] != Tile.WALL && !visited.contains(i)) {
                    addWallIfNecessary(x, y, visited);
                }

                i++;
            }
        }

        surroundByWallIfNecessary();
    }

    private void addWallIfNecessary(int x, int y, Set<Integer> visited) {
        boolean needWall = true;

        Set<Integer> localVisited = new HashSet<>();
        Stack<Integer> toVisit = new Stack<>();
        toVisit.add(y * width + x);
        localVisited.add(toVisit.peek());

        while (!toVisit.isEmpty()) {
            int i = toVisit.pop();

            int x2 = i % width;
            int y2 = i / width;

            if (x2 == playerX && y2 == playerY) {
                needWall = false;
            }

            for (Direction d : Direction.VALUES) {
                int x3 = x2 + d.dirX();
                int y3 = y2 + d.dirY();

                if (x3 < 0 || x3 >= width || y3 < 0 || y3 >= height) {
                    continue;
                }

                int i3 = y3 * width + x3;

                if (board[y3][x3] != Tile.WALL && localVisited.add(i3)) {
                    visited.add(i3);
                    toVisit.push(i3);
                }
            }
        }

        if (needWall) {
            for (Integer i : localVisited) {
                int x2 = i % width;
                int y2 = i / width;

                board[y2][x2] = Tile.WALL;
            }
        }
    }

    private void surroundByWallIfNecessary() {
        int left = 0;
        int right = 0;
        int top = 0;
        int bottom = 0;

        for (int y = 0; y < height; y++) {
            if (board[y][0] != Tile.WALL) {
                left = 1;
            }
            if (board[y][width - 1] != Tile.WALL) {
                right = 1;
            }
        }

        for (int x = 0; x < width; x++) {
            if (board[0][x] != Tile.WALL) {
                top = 1;
            }
            if (board[height - 1][x] != Tile.WALL) {
                bottom = 1;
            }
        }

        if (left == 0 && right == 0 && top == 0 && bottom == 0) {
            return;
        }

        Tile[][] newTiles = new Tile[height + top + bottom][width + right + left];

        for (int y = 0; y < height + top + bottom; y++) {
            for (int x = 0; x < width + right + left; x++) {
                if (x >= left && y >= top && x < width + left && y < height + top) {
                    newTiles[y][x] = board[y - top][x - left];
                } else {
                    newTiles[y][x] = Tile.WALL;
                }
            }
        }

        board = newTiles;
        width += right + left;
        height += top + bottom;
        playerX += left;
        playerY += top;
    }

    /**
     * Returns the player position on the x-axis
     *
     * @return the player position on the x-axis
     */
```

```java
        public int getPlayerX() {
            return playerX;
        }

        /**
         * Returns the player position on the y-axis
         *
         * @return the player position on the y-axis
         */
        public int getPlayerY() {
            return playerY;
        }

        /**
         * Set the player position to (x, y)
         *
         * @param x player position on the x-axis
         * @param y player position on the y-axis
         */
        public void setPlayerPos(int x, int y) {
            this.playerX = x;
            this.playerY = y;
        }

        /**
         * Set the player position on the x-axis to x
         *
         * @param playerX the new player position on the x-axis
         */
        public void setPlayerX(int playerX) {
            this.playerX = playerX;
        }

        /**
         * Set the player position on the y-axis to x
         *
         * @param playerY the new player position on the y-axis
         */
        public void setPlayerY(int playerY) {
            this.playerY = playerY;
        }

        private void resizeIfNeeded(int minWidth, int minHeight) {
            setSize(Math.max(minWidth, width),
                    Math.max(minHeight, height));
        }

        /**
         * Resize this level to (newWidth, newHeight). If dimensions are higher than the old one,
         * new tiles are filled with WALL. For other, tiles are the same.
         *
         * @param newWidth the new width of the level
         * @param newHeight the new width of the level
         */
        public void setSize(int newWidth, int newHeight) {
            if (newWidth == width && newHeight == height) {
                return;
            }

            Tile[][] newBoard = new Tile[newHeight][newWidth];

            int yMax = Math.min(newHeight, height);
            int xMax = Math.min(newWidth, width);
            for (int y = 0; y < yMax; y++) {
                System.arraycopy(board[y], 0, newBoard[y], 0, xMax);

                for (int x = xMax; x < newWidth; x++) {
                    newBoard[y][x] = Tile.WALL;
                }
            }

            board = newBoard;

            width = newWidth;
            height = newHeight;
        }

        /**
         * Returns the width of the level
         *
         * @return the width of the level
         */
        public int getWidth() {
            return width;
        }

        /**
         * Sets the width of the level
         *
         * @param width the new width of the level
         * @see #setSize(int, int)
         */
        public void setWidth(int width) {
            setSize(width, height);
        }

        /**
         * Returns the height of the level
         *
         * @return the height of the level
         */
        public int getHeight() {
            return height;
        }

        /**
         * Sets the height of the level
         *
         * @param height the new height of the level
         * @see #setSize(int, int)
         */
        public void setHeight(int height) {
            setSize(width, height);
        }

        /**
         * Set at (x, y) the tile. If (x, y) is outside the level, the level is resized
         *
         * @param tile the new tile
         * @param x x position
         * @param y y position
         */
        public void set(Tile tile, int x, int y) {
            resizeIfNeeded(x, y);
            board[y][x] = tile;
        }

        /**
         * Returns the tile at (x, y)
         * @param x x position of the tile
         * @param y y position of the tile
         * @return the tile at (x, y)
         */
        public Tile get(int x, int y) {
            if (x < 0 || x >= width || y < 0 || y >= height) {
                return null;
            }

            return board[y][x];
        }

        /**
         * Returns the index of the level
         * @return the index of the level
         */
        public int getIndex() {
            return index;
        }

        /**
         * Sets the index of the level
         * @param index the new index of the level
         */
        public void setIndex(int index) {
            this.index = index;
        }
    }
}
```

### SolverReport

```java
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.JsonException;
import fr.poulpogaz.json.JsonPrettyWriter;
import fr.poulpogaz.json.JsonReader;
import fr.valax.sokoshell.SokoShell;
import fr.valax.sokoshell.graphics.style.BoardStyle;
import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Move;
import fr.valax.sokoshell.solver.board.MutableBoard;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
import fr.valax.sokoshell.solver.pathfinder.Node;

import java.io.*;
import java.util.*;
import java.util.stream.Collectors;

/**
 * An object representing the output of a solver. It contains the parameters given to the solver,
 * some statistics, the solver status and if the status is {@link SolverReport#SOLUTION_FOUND},
 * it contains two representation of the solution: a sequence of {@link State} and a sequence of {@link Move}.
 *
 * @see SolverParameters
 * @see ISolverStatistics
 * @see State
 * @see Move
 * @author PoulpoGaz
 * @author darth-mole
 */
public class SolverReport {

    public static final String NO_SOLUTION = "No solution";
    public static final String SOLUTION_FOUND = "Solution found";
    public static final String STOPPED = "Stopped";
    public static final String TIMEOUT = "Timeout";
    public static final String RAM_EXCEED = "Ram exceed";

    /**
     * Creates and returns a report that doesn't contain a solution
     *
     * @param params the parameters of the solver
     * @param stats the statistics
     * @param status the solver status
     * @return a report without a solution
     * @throws IllegalArgumentException if the state is {@link SolverReport#SOLUTION_FOUND}
     */
    public static SolverReport withoutSolution(SolverParameters params, ISolverStatistics stats, String status) {
        return new SolverReport(params, stats, null, status);
    }

    /**
     * Creates and returns a report containing a solution. The solution is determined
     * from the final state.
     *
     * @param finalState the final state
     * @param params the parameters of the solver
     * @param stats the statistics
     * @return a report with a solution
     */
    public static SolverReport withSolution(State finalState, SolverParameters params, ISolverStatistics stats) {
        List<State> solution = new ArrayList<>();

        State s = finalState;
        while (s.parent() != null)
        {
            solution.add(s);
            s = s.parent();
        }
        solution.add(s);
        Collections.reverse(solution);

        return new SolverReport(params, stats, solution, SOLUTION_FOUND);
    }

    private final SolverParameters parameters;
    private final ISolverStatistics statistics;

    private final String status;

    /**
     * Solution packed in an int array.
     * Three bits are used for storing a move.
     * Move 1 is located at bit 0 of array 0,
     * Move 2 is located at bit 3 of array 0,
     * ...,
     * Move 10 is located at bit 27 of array 0,
     * Move 11 is located at bit 30 of array 0
     * and use the first bit of array 1.
     * Move 12 is located at bit 1 of array 1,
     * etc.
     * Bits are stored in little-endian fashion.
     */
    private final int[] solution;
    private final int numberOfMoves;
    private final int numberOfPushes;

    public SolverReport(SolverParameters parameters,
                        ISolverStatistics statistics,
                        List<State> states,
                        String status) {
        this.parameters = Objects.requireNonNull(parameters);
        this.statistics = Objects.requireNonNull(statistics);
        this.status = Objects.requireNonNull(status);

        if (status.equals(SOLUTION_FOUND)) {
            if (states == null) {
                throw new IllegalArgumentException("SolverStatus is SOLUTION_FOUND." +
                        "You must give the solution");
            }

            SolutionBuilder builder = createFullSolution(states);

            numberOfPushes = builder.getNumberOfPushes();
            numberOfMoves = builder.getNumberOfMoves();
            solution = builder.getSolution();
        } else {
            numberOfMoves = -1;
            numberOfPushes = -1;
            solution = null;
        }
    }

    private SolverReport(SolverParameters parameters,
                         ISolverStatistics statistics,
                         String status,
                         SolutionBuilder builder) {
        this.parameters = Objects.requireNonNull(parameters);
        this.statistics = Objects.requireNonNull(statistics);
        this.status = Objects.requireNonNull(status);

        if (status.equals(SOLUTION_FOUND)) {
            numberOfPushes = builder.getNumberOfPushes();
            numberOfMoves = builder.getNumberOfMoves();
            solution = builder.getSolution();
        } else {
            numberOfMoves = -1;
            numberOfPushes = -1;
            solution = null;
        }
    }


    /**
     * Deduce from solution's states all the moves needed to solve the sokoban
     *
     * @return the full solution
     */
    private SolutionBuilder createFullSolution(List<State> states) {
        Level level = parameters.getLevel();
        Board board = new MutableBoard(level);

        SolutionBuilder sb = new SolutionBuilder(2 * states.size());
        List<Move> temp = new ArrayList<>();

        TileInfo player = board.getAt(level.getPlayerX(), level.getPlayerY());

        CrateAStar aStar = new CrateAStar(board);
        for (int i = 0; i < states.size() - 1; i++) {
            State current = states.get(i);

            if (i != 0) {
                board.addStateCrates(current);
            }

            State next = states.get(i + 1);
            StateDiff diff = getStateDiff(board, current, next);

            Node node = aStar.findPathAndComputeMoves(
                    player, null,
                    diff.crate(), diff.crateDest());

            if (node == null) {
                throw cannotFindPathException(board, current, next);
            }

            player = node.getPlayer();
            while (node.getParent() != null) {
                temp.add(node.getMove());
                node = node.getParent();
            }

            sb.ensureCapacity(sb.getNumberOfMoves() + temp.size());
            for (int j = temp.size() - 1; j >= 0; j--) {
                sb.add(temp.get(j));
            }
            temp.clear();

            board.removeStateCrates(current);
        }

        return sb;
    }

    /**
     * Find the differences between two states:
     * <ul>
     *     <li>new player position</li>
     *     <li>old crate pos</li>
     *     <li>new crate pos</li>
     * </ul>
     *
     * @param board the board
     * @param from the first state
     * @param to the second state
     * @return a {@link StateDiff}
     */
    private StateDiff getStateDiff(Board board, State from, State to) {
        List<Integer> state1Crates =
        ↪   Arrays.stream(from.cratesIndices()).boxed().collect(Collectors.toList());
        List<Integer> state2Crates =
        ↪   Arrays.stream(to.cratesIndices()).boxed().collect(Collectors.toList());

        List<Integer> state1Copy = state1Crates.stream().toList();
        state1Crates.removeAll(state2Crates);
        state2Crates.removeAll(state1Copy);

        return new StateDiff(
                board.getAt(to.playerPos()),
                board.getAt(state1Crates.get(0)),  // original crate pos
                board.getAt(state2Crates.get(0))); // where it goes
    }

    /**
     * Create an exception indicating a path can't be found between two states.
     *
     * @param board the board which must be in the same state as current
     * @param current the current state
     * @param next the next state
     * @return an exception
     */
    private IllegalStateException cannotFindPathException(Board board, State current,
    ↪   State next) {
        BoardStyle style = SokoShell.INSTANCE.getBoardStyle();

        String str1 = style.drawToString(board, board.getX(current.playerPos()),
        ↪   board.getY(current.playerPos())).toAnsi();
        board.removeStateCrates(current);
        board.addStateCrates(next);
        String str2 = style.drawToString(board, board.getX(next.playerPos()),
        ↪   board.getY(next.playerPos())).toAnsi();

        return new IllegalStateException("""
                Can't find path between two states:
                %s
                (%s)
                and
                %s
                (%s)
                """.formatted(str1, current, str2, next));
    }




    public void writeSolution(JsonPrettyWriter jpw) throws JsonException, IOException {
        jpw.field("status", status);
        jpw.key("parameters");
        parameters.append(jpw);

        if (solution != null) {
            jpw.key("solution").beginArray();
            jpw.setInline(JsonPrettyWriter.Inline.ALL);

            for (Move m : getFullSolution()) {
                jpw.value(m.shortName());
            }

            jpw.endArray();
            jpw.setInline(JsonPrettyWriter.Inline.NONE);
        }

        jpw.key("statistics");

        // probably not a good way to do that, but I don't know
        // how to easily serialize and deserialize ISolverStatistics
        // without having a factory...
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(statistics);
        oos.close();

        jpw.value(Base64.getEncoder().encodeToString(baos.toByteArray()));
    }


    public static SolverReport fromJson(JsonReader jr, Level level) throws JsonException,
    ↪   IOException {
        String status = jr.assertKeyEquals("status").nextString();

        jr.assertKeyEquals("parameters");
        SolverParameters parameters = SolverParameters.fromJson(jr, level);

        String key = jr.nextKey();

        SolutionBuilder sb = null;
        if (key.equals("solution")) {
            jr.beginArray();

            sb = new SolutionBuilder(32 * 5); // uses array of size 16
            while (!jr.isArrayEnd()) {
                String name = jr.nextString();
                Move move = Move.of(name);

                if (move == null) {
                    throw new IOException("Unknown move: " + name);
                }

                sb.add(move);
            }
            jr.endArray();

            jr.assertKeyEquals("statistics");
        } else if (!key.equals("statistics")) {
            throw new JsonException(String.format("Invalid key. " +
                    "Expected \"statistics\" but was \"%s\"", key));
        }

        // see writeSolution
        byte[] bytes = Base64.getDecoder().decode(jr.nextString());

        ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));
        ISolverStatistics stats;
        try {
            stats = (ISolverStatistics) ois.readObject();
        } catch (ClassNotFoundException e) {
            throw new IOException(e);
```

```java
        }
        ois.close();

        return new SolverReport(parameters, stats, status, sb);
    }


    /**
     * Returns the type of the solver used to produce this report
     *
     * @return the type of the solver used to produce this report
     */
    public String getSolverName() {
        return parameters.getSolverName();
    }

    /**
     * Returns the parameters given to the solver that produce this report
     *
     * @return the parameters given to the solver
     */
    public SolverParameters getParameters() {
        return parameters;
    }

    /**
     * Returns the statistics produce by the solver that produce this report.
     * However, {@linkplain Solver solvers} are only capable of recording when
     * the research start and end. Others statistics are produced by {@link Tracker}
     *
     * @return the parameters given to the solver
     */
    public ISolverStatistics getStatistics() {
        return statistics;
    }

    public SolutionIterator getSolutionIterator() {
        if (solution == null) {
            return null;
        }

        return new SolutionIterator();
    }

    /**
     * If the sokoban was solved, this report contains the solution as a sequence
     * of moves. It describes all moves made by the player.
     *
     * @return the solution or {@code null} if the sokoban wasn't solved
     */
    public List<Move> getFullSolution() {
        if (solution == null) {
            return null;
        }

        ListIterator<Move> it = getSolutionIterator();
        List<Move> moves = new ArrayList<>(numberOfMoves);

        while (it.hasNext()) {
            moves.add(it.next());
        }

        return moves;
    }

    /**
     * Returns the number of pushes the player made to solve the sokoban
     *
     * @return {@code -1} if the sokoban wasn't solved or the number of pushes the player made to solve the sokoban
     */
    public int numberOfPushes() {
        return numberOfPushes;
    }

    /**
     * Returns the number of moves the player made to solve the sokoban
     *
     * @return {@code -1} if the sokoban wasn't solved or the number of moves the player made to solve the sokoban
     */
    public int numberOfMoves() {
        return numberOfMoves;
    }


    /**
     * Returns {@code true} if this report contains a solution
     *
     * @return {@code true} if this report contains a solution
     */
    public boolean isSolved() {
        return status.equals(SOLUTION_FOUND);
    }

    /**
     * Returns {@code true} if this report doesn't contain a solution
     *
     * @return {@code true} if this report doesn't contain a solution
     */
    public boolean hasNoSolution() {
        return !status.equals(SOLUTION_FOUND);
    }

    /**
     * Returns {@code true} if the solver was stopped by the user
     *
     * @return {@code true} if the solver was stopped by the user
     */
    public boolean isStopped() {
        return status.equals(STOPPED);
    }


    public String getStatus() {
        return status;
    }

    /**
     * Returns the level that was given to the solver
     *
     * @return the level that was given to the solver
     */
    public Level getLevel() {
        return parameters.getLevel();
    }


    /**
     * Returns the pack of the level that was given to the solver
     *
     * @return the pack of the level that was given to the solver
     */
    public Pack getPack() {
        return parameters.getLevel().getPack();
    }

    /**
     * Contains all differences between two states except the old player position.
     *
     * @param playerDest player destination
     * @param crate old crate position
     * @param crateDest crate destination
     */
    private record StateDiff(TileInfo playerDest, TileInfo crate, TileInfo crateDest) {}

    /**
     * An object to iterate over a solution in forward and backward order.
     */
    public class SolutionIterator implements ListIterator<Move> {

        /**
         * Position in the array
         */
        private int arrayPos;

        /**
         * Position in solution[arrayPos]
         */
        private int bitPos;

        private int move;
        private int push;

        /**
         * @return read the next bit
         */
        private int readNext() {
            int bit = (solution[arrayPos] >> bitPos) & 0b1;

            bitPos++;
            if (bitPos == 32) {
                bitPos = 0;
                arrayPos++;
            }

            return bit;
        }

        /**
         * @return read the previous bit
         */
        private int readPrevious() {
            bitPos--;
            if (bitPos < 0) {
                bitPos = 31;
                arrayPos--;
            }

            return (solution[arrayPos] >> bitPos) & 0b1;
        }


        @Override
        public boolean hasNext() {
            return move < numberOfMoves;
        }

        @Override
        public Move next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }

            int first  = readNext();
            int second = readNext();
            int third  = readNext();

            int value = (third << 2) | (second << 1) | first;

            Move move = Move.values()[value];

            this.move++;
            if (move.moveCrate()) {
                push++;
            }

            return move;
        }

        @Override
        public boolean hasPrevious() {
            return move > 0;
        }

        @Override
        public Move previous() {
            if (!hasPrevious()) {
                throw new NoSuchElementException();
            }

            int third  = readPrevious();
            int second = readPrevious();
            int first  = readPrevious();

            int value = (third << 2) | (second << 1) | first;

            Move move = Move.values()[value];

            this.move--;
            if (move.moveCrate()) {
                push--;
            }

            return move;
        }

        @Override
```

```java
        public int nextIndex() {
            return move;
        }

        @Override
        public int previousIndex() {
            return move - 1;
        }

        public void reset() {
            move = 0;
            arrayPos = 0;
            bitPos = 0;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }

        @Override
        public void set(Move move) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void add(Move move) {
            throw new UnsupportedOperationException();
        }

        public int getMoveCount() {
            return move;
        }

        public int getPushCount() {
            return push;
        }
    }

    /**
     * A convenience object to convert a list of move to a solution array.
     */
    private static class SolutionBuilder {

        private int[] solution;

        private int arrayPos;
        private int bitPos;

        private int numberOfMoves;
        private int numberOfPushes;

        public SolutionBuilder(int estimatedNumberOfMove) {
            solution = new int[computeArraySize(estimatedNumberOfMove)];
        }

        private void write(int bit) {
            solution[arrayPos] = (bit & 0b1) << bitPos | solution[arrayPos];

            bitPos++;
            if (bitPos == 32) {
                bitPos = 0;
                arrayPos++;
            }
        }

        public void add(Move move) {
            if (bitPos + 3 >= 32 && arrayPos + 1 >= solution.length) {
                ensureCapacity(numberOfMoves * 2 + 1);
            }

            int value = move.ordinal();
            write(value & 0b1);
            write((value >> 1) & 0b1);
            write((value >> 2) & 0b1);
            numberOfMoves++;

            if (move.moveCrate()) {
                numberOfPushes++;
            }
        }

        public void ensureCapacity(int numberOfMove) {
            int minArraySize = computeArraySize(numberOfMove);

            if (minArraySize > solution.length) {
                solution = Arrays.copyOf(solution, minArraySize);
            }
        }

        public int getNumberOfMoves() {
            return numberOfMoves;
        }

        public int getNumberOfPushes() {
            return numberOfPushes;
        }

        public int[] getSolution() {
            int arraySize = computeArraySize(numberOfMoves);

            return Arrays.copyOf(solution, arraySize);
        }

        private int computeArraySize(int numberOfMove) {
            int nBits = 3 * numberOfMove;

            return nBits / 32 + 1;
        }
    }
}
```

### FESS0Solver

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverCollection;
import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
import fr.valax.sokoshell.solver.heuristic.Heuristic;

import java.util.PriorityQueue;

```

```
public class FESS0Solver extends AbstractSolver<FESS0Solver.FESS0State> {

    private Heuristic heuristic;
    private int lowerBound;

    public FESS0Solver() {
        super("fess0");
    }

    @Override
    protected void init(SolverParameters parameters) {
        heuristic = new GreedyHeuristic(board);
        toProcess = new SolverPriorityQueue();
    }

    @Override
    protected void addInitialState(Level level) {
        CorralDetector detector = board.getCorralDetector();
        State s = level.getInitialState();

        board.addStateCrates(s);
        detector.findCorral(board, s.playerPos() % level.getWidth(), s.playerPos() /
        ↪   level.getWidth());
        board.removeStateCrates(s);

        lowerBound = heuristic.compute(s);

        FESS0State state = new FESS0State(s, 0, lowerBound,
        ↪   detector.getRealNumberOfCorral(), countPackedCrate(s));

        toProcess.addState(state);
    }

    @Override
    protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
        if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
            return;
        }

        final int i = board.topLeftReachablePosition(crate, crateDest);
        // The new player position is the crate position
        FESS0State s = toProcess.cachedState().child(i, crate.getCrateIndex(),
        ↪   crateDest.getIndex());
        s.setHeuristic(heuristic.compute(s));
        s.setConnectivity(board.getCorralDetector().getRealNumberOfCorral());
        s.setPacking(countPackedCrate(s));

        if (processed.add(s)) {
            toProcess.addState(s);
        }
    }

    private int countPackedCrate(State state) {
        int nPacked = 0;
        for (int crate : state.cratesIndices()) {
            TileInfo tile = board.getAt(crate);
            if (tile.isTarget() || tile.isCrateOnTarget()) {
                nPacked++;
            }
        }

        return nPacked;
```

```java
    }

    @Override
    public int lowerBound() {
        return lowerBound;
    }

    private static class SolverPriorityQueue extends PriorityQueue<FESS0State>
            implements SolverCollection<FESS0State> {

        private FESS0State cachedState;

        @Override
        public void addState(FESS0State state) {
            offer(state);
        }

        @Override
        public FESS0State popState() {
            return poll();
        }

        @Override
        public FESS0State peekState() {
            return peek();
        }

        @Override
        public FESS0State peekAndCacheState() {
            cachedState = popState();
            return cachedState;
        }

        @Override
        public FESS0State cachedState() {
            return cachedState;
        }
    }

    protected static class FESS0State extends WeightedState implements
    ↪   Comparable<FESS0State> {

        private int connectivity;
        private int packing;

        public FESS0State(int playerPos, int[] cratesIndices, int hash, State parent, int
        ↪   cost, int heuristic) {
            super(playerPos, cratesIndices, hash, parent, cost, heuristic);
        }

        public FESS0State(State state, int cost, int heuristic, int connectivity, int
        ↪   packing) {
            super(state, cost, heuristic);
            this.connectivity = connectivity;
            this.packing = packing;
        }

        @Override
        public FESS0State child(int newPlayerPos, int crateToMove, int crateDestination) {
            return new FESS0State(super.child(newPlayerPos, crateToMove,
            ↪   crateDestination),
                    cost(), 0, 0, 0);
        }

        public int getConnectivity() {
            return connectivity;
        }

        public void setConnectivity(int connectivity) {
            this.connectivity = connectivity;
        }

        public int getPacking() {
            return packing;
        }

        public void setPacking(int packing) {
            this.packing = packing;
        }

        @Override
        public int compareTo(FESSOState o) {
            // compare in reverse order because
            // java PriorityQueue is a min-queue
            return compare(o, this);
        }

        private static int compare(FESSOState a, FESSOState b) {
            // -1 if this < o
            //  0 if this = o
            //  1 if this > o
            if (a.packing > b.packing) {
                return 1; // we want to maximize packing
            } else if (a.packing < b.packing) {
                return -1;
            } else {
                if (a.connectivity < b.connectivity) {
                    return 1; // we want to minimize connectivity
                } else if (a.connectivity > b.connectivity) {
                    return -1;
                } else {
                    return Integer.compare(a.weight(), b.weight());
                }
            }
        }
    }
}
```

### SolverParameters

```java
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.IJsonReader;
import fr.poulpogaz.json.IJsonWriter;
import fr.poulpogaz.json.JsonException;
import fr.valax.sokoshell.SokoShell;

import java.io.IOException;
import java.util.*;

/**
 * A collection of {@link SolverParameter} plus the name of the solver used and the level
↪  to
 * solve. {@link Solver} known which level to solve thanks to this object
 */
public class SolverParameters {

    private final String solverName;
    private final Level level;
    private final Map<String, SolverParameter> parameters;

    public SolverParameters(String solverName, Level level) {
        this(solverName, level, null);
    }

    public SolverParameters(String solverName, Level level, List<SolverParameter> parameters) {
        this.solverName = Objects.requireNonNull(solverName);
        this.level = Objects.requireNonNull(level);

        if (parameters == null) {
            this.parameters = Map.of();
        } else {
            this.parameters = new HashMap<>();

            for (SolverParameter p : parameters) {
                this.parameters.put(p.getName(), p);
            }
        }
    }

    /**
     * @param param parameter name
     * @return the parameter named param
     */
    public SolverParameter get(String param) {
        return parameters.get(param);
    }

    /**
     *
     * @param param name of the parameter
     * @return argument of parameter param or default value
     * @param <T> type of the argument
     * @throws ClassCastException if the argument can't be cast to a T
     */
    @SuppressWarnings("unchecked")
    public <T> T getArgument(String param) {
        SolverParameter p = parameters.get(param);

        if (p == null) {
            throw new NoSuchElementException("No such parameter: " + param);
        }

        return (T) p.getOrDefault();
    }

    /**
     * @return all parameters
     */
    public Collection<SolverParameter> getParameters() {
        return parameters.values();
    }

    /**
     * @return the level to solve
     */
    public Level getLevel() {
        return level;
    }

    /**
     * @return the name of the solver used
     */
    public String getSolverName() {
        return solverName;
    }


    public void append(IJsonWriter jw) throws JsonException, IOException {
        jw.beginObject();
        jw.field("solver", solverName);

        for (Map.Entry<String, SolverParameter> param : parameters.entrySet()) {
            if (param.getValue().hasArgument()) {
                jw.key(param.getKey());
                param.getValue().toJson(jw);
            }
        }

        jw.endObject();
    }

    public static SolverParameters fromJson(IJsonReader jr, Level level) throws
     ↪  JsonException, IOException {
        jr.beginObject();
        String solverName = jr.assertKeyEquals("solver").nextString();

        Solver solver = SokoShell.INSTANCE.getSolver(solverName);
        if (solver == null) {
            throw new IOException("No such solver: " + solverName);
        }

        List<SolverParameter> parameters = solver.getParameters();
        while (!jr.isObjectEnd()) {
            String key = jr.nextKey();

            SolverParameter parameter = parameters.stream()
                    .filter((s) -> s.getName().equals(key))
                    .findFirst()
                    .orElseThrow(() -> new IOException("No such parameter: " + key));

            parameter.fromJson(jr);
        }

        jr.endObject();

        return new SolverParameters(solverName, level, parameters);
    }
}
```

### FreezeDeadlockDetector

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.Tile;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

public class FreezeDeadlockDetector {

    // http://www.sokobano.de/wiki/index.php?title=How_to_detect_deadlocks
    public static boolean checkFreezeDeadlock(Board board, State state) {
        int[] crates = state.cratesIndices();

        for (int crate : crates) {
            TileInfo info = board.getAt(crate);

            if (checkFreezeDeadlock(info)) {
                return true;
            }
        }

        return false;
    }

    public static boolean checkFreezeDeadlock(TileInfo crate) {
        return crate.isCrate() &&
                checkFreezeDeadlockRec(crate, Direction.LEFT) &&
                checkFreezeDeadlockRec(crate, Direction.UP);
    }

    private static boolean checkFreezeDeadlockRec(TileInfo crate) {
        return checkFreezeDeadlockRec(crate, Direction.LEFT) &&
                checkFreezeDeadlockRec(crate, Direction.UP);
    }

    private static boolean checkFreezeDeadlockRec(TileInfo current, Direction axis) {
        boolean deadlock = false;

        TileInfo left = current.adjacent(axis);
        TileInfo right = current.adjacent(axis.negate());

        if (left.isWall() || right.isWall()) { // rule 1
            deadlock = true;

        } else if (left.isDeadTile() && right.isDeadTile()) { // rule 2
            deadlock = true;

        } else { // rule 3
            Tile oldCurr = current.getTile();
            current.setTile(Tile.WALL);

            if (left.anyCrate()) {
                deadlock = checkFreezeDeadlockRec(left);
            }

            if (!deadlock && right.anyCrate()) {
                deadlock = checkFreezeDeadlockRec(right);
            }

            current.setTile(oldCurr);
        }

        return deadlock;
    }
}
```

**ISolverStatistics**

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.utils.PrettyTable;
import fr.valax.sokoshell.utils.Utils;

import java.io.PrintStream;
import java.io.Serializable;

/**
 * An object that contains various statistics about a solution, including
 * time start and end, number of node explored and queue size at a specific instant
 */
public interface ISolverStatistics extends Serializable {

    /**
     * Returns the time in millis when the solver was started
     *
     * @return the time in millis when the solver was started
     */
    long timeStarted();

    /**
     * Returns the time in millis when the solver stopped running
     *
     * @return the time in millis when the solver stopped running
     */
    long timeEnded();

    /**
     * Returns the time used by the solver to solve a level
     *
     * @return the run time in millis
     */
    default long runTime() {
        return timeEnded() - timeStarted();
    }

    /**
     * Returns the total number of state explored by the solver.
     * If the solver doesn't use State or the {@link Tracker}
     * doesn't compute this property, implementations can return
     * a negative number
     *
     * @return total number of state explored
     */
    int totalStateExplored();

    /**
     * @return number of state explored per seconds or -1
     */
    long stateExploredPerSeconds();

    /**
     * @return average queue size or -1
     */
    int averageQueueSize();

    /**
     * @return lower bound or -1
     */
    int lowerBound();

    /**
     * Print statistics to out.
     *
     * @param out standard output stream
     * @param err error output stream
     * @return an optional table containing statistics
     */
    default PrettyTable printStatistics(PrintStream out, PrintStream err) {
        out.printf("Started at %s. Finished at %s. Run time: %s%n",
                Utils.formatDate(timeStarted()),
                Utils.formatDate(timeEnded()),
                Utils.prettyDate(runTime()));

        return null;
    }

    /**
     * Basic implementation of {@link ISolverStatistics} then just
     * save time started and time ended
     */
    record Basic(long timeStarted, long timeEnded) implements ISolverStatistics {

        @Override
        public int totalStateExplored() {
            return -1;
        }

        @Override
        public long stateExploredPerSeconds() {
            return -1;
        }

        @Override
        public int averageQueueSize() {
            return -1;
        }

        @Override
        public int lowerBound() {
            return -1;
        }
    }
}
```

### Pack

```
package fr.valax.sokoshell.solver;

import fr.poulpogaz.json.JsonException;
import fr.poulpogaz.json.JsonPrettyWriter;
import fr.poulpogaz.json.JsonReader;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Collections;
import java.util.List;
import java.util.Objects;
import java.util.zip.GZIPInputStream;
```

```java
import java.util.zip.GZIPOutputStream;

/**
 * A pack is a collection of levels with a name and an author
 */
public final class Pack {

    /**
     * Some pack doesn't have a name while it is required by {@link fr.valax.sokoshell.SokoShell}.
     * So pack without a name as named as following: 'Unnamed[I]' where I is an integer which is increased
     * each time an unnamed pack is created. This 'I' is the variable below
     */
    private static int unnamedIndex = 0;


    private final String name;
    private final String author;
    private final List<Level> levels;

    private Path sourcePath;

    public Pack(String name, String author, List<Level> levels) {
        if (name == null) {
            this.name = "Unnamed[" + unnamedIndex + "]";
            unnamedIndex++;
        } else {
            this.name = name;
        }

        this.author = author;
        this.levels = Collections.unmodifiableList(levels);

        for (Level level : this.levels) {
            level.pack = this;
        }
    }

    public void writeSolutions(Path out) throws IOException, JsonException {
        if (out == null) {
            out = Path.of(sourcePath.toString() + ".solutions.json.gz");
        }

        boolean write = false;
        for (Level level : levels) {
            if (level.hasReport()) {
                write = true;
            }
        }

        if (!write) {
            return;
        }

        try (OutputStream os = Files.newOutputStream(out, StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING)) {
            BufferedWriter bw = new BufferedWriter(
                    new OutputStreamWriter(new GZIPOutputStream(os)));

            JsonPrettyWriter jpw = new JsonPrettyWriter(bw);

            jpw.beginObject();
            if (name != null) {
                jpw.field("pack", name);
            } else {
                jpw.nullField("pack");
            }
            if (author != null) {
                jpw.field("author", author);
            } else {
                jpw.nullField("author");
            }

            for (Level level : levels) {
                if (level.hasReport()) {

                    jpw.key(String.valueOf(level.getIndex()));
                    jpw.beginArray();

                    level.writeSolutions(jpw);

                    jpw.endArray();
                }
            }

            jpw.endObject();
            jpw.close();
        }
    }

    public void readSolutions(Path in) throws IOException, JsonException {
        if (Files.notExists(in)) {
            return;
        }

        try (InputStream is = Files.newInputStream(in)) {
            BufferedReader br = new BufferedReader(
                    new InputStreamReader(new GZIPInputStream(is)));

            JsonReader jr = new JsonReader(br);

            jr.beginObject();
            jr.assertKeyEquals("pack");

            String pack;
            if (jr.hasNextString()) {
                pack = jr.nextString();
            } else {
                jr.nextNull();
                pack = null;
            }

            jr.assertKeyEquals("author");
            String author;
            if (jr.hasNextString()) {
                author = jr.nextString();
            } else {
                jr.nextNull();
                author = null;
            }

            if (Objects.equals(pack, name) && Objects.equals(author, this.author)) {

```

```java
                while (!jr.isObjectEnd()) {
                    int level = Integer.parseInt(jr.nextKey());

                    Level l = levels.get(level);
                    jr.beginArray();

                    while (!jr.isArrayEnd()) {
                        jr.beginObject();
                        l.addSolverReport(SolverReport.fromJson(jr, l));
                        jr.endObject();
                    }

                    jr.endArray();
                }

                jr.endObject();
            }
            jr.close();
        }
    }

    /**
     * Returns the name of the pack
     *
     * @return the name of the pack
     */
    public String name() {
        return name;
    }

    /**
     * Returns the author of the pack
     *
     * @return pack's author
     */
    public String author() {
        return author;
    }

    /**
     * Returns all levels that are in this pack
     *
     * @return levels of this pack
     */
    public List<Level> levels() {
        return levels;
    }

    /**
     * Returns the level at the specified index
     *
     * @param index the index of the level
     * @return the level at the specified index
     * @throws IndexOutOfBoundsException if the index is out of range
     */
    public Level getLevel(int index) {
        return levels.get(index);
    }

    /**
     * Returns the number of level in this pack
     *
     * @return the number of level in this pack
     */
    public int nLevels() {
        return levels.size();
    }

    /**
     * Returns the location of the file describing this pack. This is used for writing solutions
     *
     * @return the location of the file describing this pack
     * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
     */
    public Path getSourcePath() {
        return sourcePath;
    }

    /**
     * Sets the location of the file describing this pack. This is used for writing solutions
     *
     * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
     */
    public void setSourcePath(Path sourcePath) {
        this.sourcePath = sourcePath;
    }
}
```

### Hotspots

```java
package fr.valax.sokoshell.solver;

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;

import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Queue;

public class Hotspots {

    private final Board board;

    /**
     * if hotspot[X][Y] is true then if there is a crate at Y and at X,
     * Y blocks X to be pushed to at least one target
     */
    private final boolean[][] hotspot;

    // Variables used by countAccessibleTargets
    private ReachableTiles reachable;
    // accessible[x] is true if at x there is target and the target is push-accessible
    private boolean[] accessible;
    private Queue<State> toVisit;
    private HashSet<State> visited;

    public Hotspots(Board board) {
        this.board = board;
        int s = board.getWidth() * board.getHeight();
        this.hotspot = new boolean[s][s];
    }

    protected void postInit() {
        int size = board.getWidth() * board.getHeight();
        reachable = new ReachableTiles(board);
        accessible = new boolean[size];
        toVisit = new ArrayDeque<>();
        visited = new HashSet<>();
    }

    public void computeHotspots() {
        postInit();
        int size = board.getWidth() * board.getHeight();

        for (int X = 0; X < size; X++) {
            TileInfo x = board.getAt(X);
            if (x.isSolid()) {
                continue;
            }

            x.addCrate();
            int accessible = countAccessibleTargets(board, X);

            for (int Y = 0; Y < size; Y++) {
                if (Y == X) {
                    continue;
                }

                TileInfo y = board.getAt(Y);
                if (y.isSolid()) {
                    continue;
                }

                y.addCrate();
                int accessible2 = countAccessibleTargets(board, X);
                y.removeCrate();

                if (accessible != accessible2) {
                    hotspot[X][Y] = true;
                }
            }
            x.removeCrate();
        }

        reachable = null;
        accessible = null;
        toVisit = null;
        visited = null;
    }

    protected int countAccessibleTargets(Board board, int baseCratePos) {
        Arrays.fill(accessible, false);
        toVisit.clear();
        visited.clear();

        // add base state
        // There is four state, for each direction
        TileInfo baseCrate = board.getAt(baseCratePos);
        for (Direction d : Direction.VALUES) {
            TileInfo player = baseCrate.adjacent(d);

            if (!player.isSolid()) {
```

```java
                State s = new State(player, baseCrate);
                toVisit.add(s);
                visited.add(s);
            }
        }

        if (baseCrate.isCrateOnTarget()) {
            accessible[baseCratePos] = true;
        }

        baseCrate.removeCrate();

        while (!toVisit.isEmpty()) {
            State s = toVisit.poll();

            s.crate().addCrate();
            reachable.findReachableCases(s.player());

            for (Direction dir : Direction.VALUES) {
                TileInfo player = s.crate().adjacent(dir.negate());
                TileInfo crateDest = s.crate().adjacent(dir);

                if (crateDest.isSolid() || !reachable.isReachable(player)) {
                    continue;
                }

                if (crateDest.isTarget()) {
                    accessible[crateDest.getIndex()] = true;
                }

                int playerDest = board.topLeftReachablePosition(s.crate(), crateDest);

                State newState = new State(board.getAt(playerDest), crateDest);
                if (visited.add(newState)) {
                    toVisit.add(newState);
                }
            }

            s.crate().removeCrate();
        }
        baseCrate.addCrate();

        return countTrue(accessible);
    }

    private int countTrue(boolean[] array) {
        int n = 0;

        for (int i = 0; i < array.length; i++) {
            if (array[i]) {
                n++;
            }
        }

        return n;
    }

    public boolean isHotspot(int crate, int blockingCrate) {
        return hotspot[crate][blockingCrate];
    }

    private record State(TileInfo player, TileInfo crate) {}
```

```
}
```