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);
```

```
58
59          if (dest.isSolid()) {
60              return null;
61          }
62
63          int i = toIndex(dest);
64          Node node = nodes[i];
65
66          if (markSystem.isMarked(i) || markSystem.isVisited(i)) { // the node was added to the queue,
            ↪  therefore node.getExpectedDist() is valid
67              if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {
68                  node.changeParent(parent);
69                  decreasePriority(node);
70              }
71
72              return null;
73          } else {
74              markSystem.mark(i);
75              node.set(parent, dest, null, heuristic(dest));
76              return node;
77          }
78      }
79
80      @Override
81      protected void markVisited(Node node) {
82          markSystem.setVisited(toIndex(node.getPlayer()));
83      }
84
85      @Override
86      protected boolean isVisited(Node node) {
87          return markSystem.isVisited(toIndex(node.getPlayer()));
88      }
89
90      protected int heuristic(TileInfo newPlayer) {
91          return newPlayer.manhattanDistance(playerDest);
92      }
93
94      @Override
95      protected boolean isEndNode(Node node) {
96          return node.getPlayer().isAt(playerDest);
97      }
98  }
```

AStarMarkSystem

```
1  package fr.valax.sokoshell.solver.pathfinder;
2
3  import fr.valax.sokoshell.solver.board.mark.Mark;
4  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
5  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
7  /**
8   * A mark is visited, if it is equal to the global mark.
9   * A mark is marked, if it is equal to the global mark minus one.
10  * It is used because, in A*, I need to know when I first encounter
11  * a node (mark) and when I poll a node from the PriorityQueue (visited).
12  * A node which isn't marked has a wrong expected dist, inherited from a previous
13  * call to {@link AbstractAStar#findPath(TileInfo, TileInfo, TileInfo, TileInfo)}
14  */
15 public class AStarMarkSystem implements MarkSystem {
16
17      private int mark = 0;
```

```
18      private final AStarMark[] marks;
19
20      public AStarMarkSystem(int capacity) {
21          marks = new AStarMark[capacity];
22
23          for (int i = 0; i < capacity; i++) {
24              marks[i] = new AStarMark();
25          }
26      }
27
28      @Override
29      public Mark newMark() {
30          throw new UnsupportedOperationException();
31      }
32
33      /**
34       * Unmark and <strong>un-visit</strong> all mark
35       */
36      @Override
37      public void unmarkAll() {
38          mark += 2;
39      }
40
41      public void mark(int i) {
42          marks[i].mark();
43      }
44
45      public void setVisited(int i) {
46          marks[i].setVisited();
47      }
48
49      public boolean isMarked(int i) {
50          return marks[i].isMarked();
51      }
52
53      public boolean isVisited(int i) {
54          return marks[i].isVisited();
55      }
56
57      @Override
58      public void reset() {
59          mark = 0;
60
61          for (AStarMark mark : marks) {
62              mark.unmark();
63          }
64      }
65
66      @Override
67      public int getMark() {
68          return 0;
69      }
70
71      private class AStarMark implements Mark {
72
73          private int mark = AStarMarkSystem.this.mark - 2;
74
75          @Override
76          public void mark() {
77              mark = AStarMarkSystem.this.mark - 1;
78          }
79
80          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

```
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;
    }
```

```
32
33      /**
34       * It also computes the move field in {@link Node}
35       *
36       * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
37       */
38      public Node findPathAndComputeMoves(TileInfo playerStart, TileInfo playerDest, TileInfo
        ↪  crateStart, TileInfo crateDest) {
39          Node end = findPath(playerStart, playerDest, crateStart, crateDest);
40
41          if (end == null) {
42              return null;
43          }
44
45          Node current = end;
46          while (current.getParent() != null) {
47              Node last = current.getParent();
48
49              TileInfo lastPlayer = last.getPlayer();
50              TileInfo currPlayer = current.getPlayer();
51              Direction dir = Direction.of(currPlayer.getX() - lastPlayer.getX(), currPlayer.getY() -
                ↪  lastPlayer.getY());
52
53              boolean moved = crateStart != null && !current.getCrate().isAt(last.getCrate());
54              current.setMove(Move.of(dir, moved));
55
56              current = last;
57          }
58
59          return end;
60      }
61
62      /**
63       * Find a path between (playerStart, crateStart) and (playerDest, crateDest).
64       * The returned node may be cached by the implementation. Therefore, if you
65       * want to keep the path in memory, you need to copy the path.
66       *
67       * @param playerStart player start
68       * @param playerDest player dest
69       * @param crateStart crate start
70       * @param crateDest crate dest
71       * @return the shortest path as a linked list in reverse.
72       */
73      public Node findPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart, TileInfo
        ↪  crateDest) {
74          this.playerStart = playerStart;
75          this.crateStart = crateStart;
76          this.playerDest = playerDest;
77          this.crateDest = crateDest;
78
79          init();
80          Node n = initialNode();
81          queue.offer(n);
82
83          // int c = 0;
84          Node end = null;
85          while (!queue.isEmpty()) {
86              Node node = queue.poll();
87
88              if (isEndNode(node)) {
89                  end = node;
90                  break;
91              }
```

```
92
93              if (isVisited(node)) {
94                  continue;
95              }
96
97              for (Direction direction : Direction.VALUES) {
98                  Node child = processMove(node, direction);
99
100                 if (child != null) {
101                     queue.offer(child);
102                 }
103             }
104
105             markVisited(node);
106             // c++;
107         }
108         // System.out.println(c);
109
110         clean();
111         return end;
112     }
113
114     /**
115      * Decrease the priority of the node in the queue if and only if it is in the queue
116      * @param node node
117      */
118     public void decreasePriority(Node node) {
119         // TODO: we do not have a fixed size binary heap that
120         //  can efficiently decrease priority (at least O(log n))
121         if (queue.remove(node)) { // takes O(n)
122             queue.offer(node); // takes O(log n)
123         }
124     }
125
126     /**
127      * Init A*. Usually clear the queue. Called before the search
128      */
129     protected abstract void init();
130
131     /**
132      * Clean the object. Called at the end of the search
133      */
134     protected abstract void clean();
135
136     /**
137      * Returns the initial node.
138      * @return the initial node
139      */
140     protected abstract Node initialNode();
141
142     /**
143      *
144      * @param parent parent node
145      * @param dir direction taken player
146      * @return {@code null} if the player cannot move in the specified direction
147      * or if the node was already visited. Otherwise, returns child node
148      */
149     protected abstract Node processMove(Node parent, Direction dir);
150
151     /**
152      * Mark the node as visited
153      * @param node node
154      */
```

```
155     protected abstract void markVisited(Node node);
156 
157     /**
158      * @param node node
159      * @return {@code true} if the node is visited
160      */
161     protected abstract boolean isVisited(Node node);
162 
163     /**
164      * @param node node
165      * @return {@code true} if this node represents the solution
166      */
167     protected abstract boolean isEndNode(Node node);
168 }
```

CratePlayerAStar

```
1  package fr.valax.sokoshell.solver.pathfinder;
2 
3  import fr.valax.sokoshell.solver.board.Board;
4  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
5 
6  /**
7   * Find the shortest path between (player start, crate start) and (player dest, crate dest):
8   * the player moves a crate from 'crate start' to 'crate dest' and then moves to 'player dest'.
9   */
10 public class CratePlayerAStar extends CrateAStar {
11 
12     public CratePlayerAStar(Board board) {
13         super(board);
14     }
15 
16     @Override
17     protected boolean isEndNode(Node node) {
18         return node.getPlayer().isAt(playerDest) && node.getCrate().isAt(crateDest);
19     }
20 
21     @Override
22     protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
23          /*
24             Try to first move the player near the crate
25             Then push the crate to his destination
26             Finally moves the player to his destination
27          */
28         int remaining = newCrate.manhattanDistance(crateDest);
29         if (remaining == 0) {
30             remaining = newPlayer.manhattanDistance(playerDest);
31         } else {
32             if (newPlayer.manhattanDistance(newCrate) > 1) {
33                 remaining += newPlayer.manhattanDistance(newCrate);
34             }
35 
36             remaining += crateDest.manhattanDistance(playerDest);
37         }
38 
39         return remaining;
40     }
41 }
```

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;
    }

```

```
63      public Node getParent() {
64          return parent;
65      }
66
67      public int getDist() {
68          return dist;
69      }
70
71      public int getHeuristic() {
72          return heuristic;
73      }
74
75      public TileInfo getPlayer() {
76          return player;
77      }
78
79      public TileInfo getCrate() {
80          return crate;
81      }
82
83      public Move getMove() {
84          return move;
85      }
86
87      public void setMove(Move move) {
88          this.move = move;
89      }
90
91      public int getExpectedDist() {
92          return expectedDist;
93      }
94
95      @Override
96      public boolean equals(Object o) {
97          if (this == o) return true;
98          if (!(o instanceof Node node)) return false;
99
100         if (!Objects.equals(player, node.player)) return false;
101         return Objects.equals(crate, node.crate);
102     }
103
104     @Override
105     public int hashCode() {
106         int result = player != null ? player.getIndex() : 0;
107         result = 31 * result + (crate != null ? crate.getIndex() : 0); // TODO
108         return result;
109     }
110
111     @Override
112     public int compareTo(Node o) {
113         return Integer.compare(expectedDist, o.expectedDist);
114     }
115 }
```

CrateAStar

```
1  package fr.valax.sokoshell.solver.pathfinder;
2
3  import fr.valax.sokoshell.solver.board.Board;
4  import fr.valax.sokoshell.solver.board.Direction;
5  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
```

```java
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);

```

```
69              if (crateDest.isSolid()) {
70                  return null;
71              }
72
73              // check deadlock
74              if (!crateDest.isAt(this.crateDest) && // not a deadlock is if is destination
75                      crateDest.adjacent(dir).isSolid() && // front must be solid
76                      (crateDest.adjacent(dir.left()).isSolid() || // perp must be solid
77                              crateDest.adjacent(dir.right()).isSolid())) {
78                  return null;
79              }
80          } else if (playerDest.isSolid()) {
81              return null;
82          }
83
84          int i = toIndex(playerDest, crateDest);
85          Node node = nodes[i];
86
87          if (markSystem.isMarked(i) || markSystem.isVisited(i)) {
88              if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {
89                  node.changeParent(parent);
90                  decreasePriority(node);
91              }
92
93              return null;
94          } else {
95              markSystem.mark(i);
96              node.set(parent, playerDest, crateDest, heuristic(playerDest, crateDest));
97
98              return node;
99          }
100     }
101
102     @Override
103     protected void markVisited(Node node) {
104         markSystem.setVisited(toIndex(node.getPlayer(), node.getCrate()));
105     }
106
107     @Override
108     protected boolean isVisited(Node node) {
109         return markSystem.isVisited(toIndex(node.getPlayer(), node.getCrate()));
110     }
111
112     @Override
113     protected boolean isEndNode(Node node) {
114         return node.getCrate().isAt(crateDest);
115     }
116
117     protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
118         int h = newCrate.manhattanDistance(crateDest);
119
120         /* the player first need to move near the crate to push it
121            may not be optimal for level like this:
122
123             #########
124             #       #
125             # ##### #
126             # ##### #
127             # ##### #
128              @$     # The player needs to do a detour to push the crate
129             # #######
130          */
131         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) {
        this.value = value;
    }
}
```

SolverCollection

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

```

```java
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;
    }
```

```
38
39      protected int leftChild(int i) {
40          return 2 * i + 1;
41      }
42
43      protected int rightChild(int i) {
44          return 2 * i + 2;
45      }
46
47      protected void moveNodeUp(int i) {
48          if (i == 0) {
49              return;
50          }
51          final int p = parent(i);
52          if (nodes.get(i).hasPriorityOver(nodes.get(p))) {
53              Collections.swap(nodes, i, p);
54              moveNodeUp(p);
55          }
56      }
57
58      protected void moveNodeDown(int i) {
59          int j = i;
60          final int l = leftChild(i), r = rightChild(i);
61          if (l < size() && nodes.get(l).hasPriorityOver(nodes.get(i))) {
62              j = l;
63          }
64          if (r < size() && nodes.get(r).hasPriorityOver(nodes.get(l))) {
65              j = r;
66          }
67
68          if (i != j) {
69              Collections.swap(nodes, i, j);
70              moveNodeDown(j);
71          }
72      }
73
74      private int parent(int i) {
75          assert i != 0;
76          return (i - 1) / 2;
77      }
78
79      public void add(T content, int priority) {
80          int i = 0;
81          if (currentSize == -1) {
82              nodes.add(new Node<>(content, priority));
83              moveNodeUp(nodes.size() - 1);
84          } else {
85              nodes.get(currentSize).set(content, priority);
86              moveNodeUp(currentSize);
87              currentSize++;
88          }
89      }
90
91      public T pop() {
92          final int i = size() - 1;
93          Collections.swap(nodes, 0, i);
94          T content;
95          if (currentSize == -1) {
96              content = nodes.remove(i).content();
97          } else {
98              content = nodes.get(i).content();
99              currentSize--;
100         }
```

```
        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;
```

```
164         }
165
166         public void setPriority(int priority) {
167             this.priority = priority;
168         }
169     }
170 }
```

SolverPriorityQueue

```
1  package fr.valax.sokoshell.solver.collections;
2
3  import fr.valax.sokoshell.solver.WeightedState;
4
5  /**
6   * Priority queue of dynamic capacity. The priority are in <strong>ASCENDANT</strong> order, i.e. the
   ↪  element returned
7   * by {@link SolverPriorityQueue#popState()} with the <strong>LOWEST</strong> priority.
8   */
9  public class SolverPriorityQueue implements SolverCollection<WeightedState> {
10
11     /**
12      * @implNote We use a min heap collection.
13       */
14     private final MinHeap<WeightedState> heap = new MinHeap<>();
15
16     private WeightedState cachedState;
17
18     @Override
19     public void addState(WeightedState state) {
20         heap.add(state, state.weight());
21     }
22
23     @Override
24     public WeightedState popState() {
25         return heap.pop();
26     }
27
28     @Override
29     public WeightedState peekState() {
30         return heap.peek();
31     }
32
33     @Override
34     public WeightedState peekAndCacheState() {
35         cachedState = popState();
36         return cachedState;
37     }
38
39     @Override
40     public WeightedState cachedState() {
41         return cachedState;
42     }
43
44     @Override
45     public void clear() {
46         heap.clear();
47     }
48
49     @Override
50     public boolean isEmpty() {
51         return heap.isEmpty();
```

```
52      }
53
54      @Override
55      public int size() {
56          return heap.size();
57      }
58  }
59
```

1.3 heuristic

AbstractHeuristic

```
1   package fr.valax.sokoshell.solver.heuristic;
2
3   import fr.valax.sokoshell.solver.board.Board;
4
5   /**
6    * Base class for heuristic computing classes.
7    * As there are different ways to compute the heuristic of a state, we provide a set of class each
    ↪  implementing
8    * different heuristic calculation methods.
9    */
10  public abstract class AbstractHeuristic implements Heuristic {
11
12      protected final Board board;
13
14      public AbstractHeuristic(Board board) {
15          this.board = board;
16      }
17  }
```

GreedyHeuristic

```
1   package fr.valax.sokoshell.solver.heuristic;
2
3   import fr.valax.sokoshell.solver.State;
4   import fr.valax.sokoshell.solver.board.Board;
5   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
7   /**
8    * According to <a href="http://sokobano.de/wiki/index.php?title=Solver#Greedy_approach">this
    ↪  article</a>
9    */
10  public class GreedyHeuristic extends AbstractHeuristic {
11
12      private final LinkedList list;
13
14      public GreedyHeuristic(Board board) {
15          super(board);
16          final int n = board.getTargetCount();
17
18          list = new LinkedList(n);
19      }
20
21      @Override
22      public int compute(State s) {
23          int heuristic = 0;
24
25          board.getMarkSystem().unmarkAll();
```

```
26
27          int n = 0;
28          for (int crate : s.cratesIndices()) {
29              TileInfo tile = board.getAt(crate);
30
31              if (tile.isCrateOnTarget()) {
32                  tile.mark();
33              } else {
34                  list.add(tile);
35
36                  n++;
37              }
38          }
39
40
41          for (int i = 0; i < n; i++) {
42              Node minNode = list.getHead();
43              TileInfo.TargetRemoteness minDist = minNode.getNearestNotAttributedTarget();
44
45              Node node = minNode.nextNode();
46              while (node != null) {
47                  TileInfo.TargetRemoteness nearest = node.getNearestNotAttributedTarget();
48
49                  if (nearest.distance() < minDist.distance()) {
50                      minNode = node;
51                      minDist = nearest;
52                  }
53
54                  node = node.nextNode();
55              }
56
57              board.getAt(minDist.index()).mark();
58              minNode.getCrate().mark();
59              heuristic += minDist.distance();
60
61              minNode.remove();
62          }
63
64          return heuristic;
65      }
66
67      private static class LinkedList {
68
69          private final Node[] nodeCache;
70          private int size = 0;
71
72          private Node head;
73
74          public LinkedList(int size) {
75              nodeCache = new Node[size];
76
77              for (int i = 0; i < size; i++) {
78                  nodeCache[i] = new Node(this);
79              }
80          }
81
82          public void add(TileInfo crate) {
83              Node newHead = nodeCache[size];
84              newHead.set(crate);
85
86              if (head != null) {
87                  newHead.next = head;
88                  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() {
            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

```
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

```
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

```java
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;

    /**
     * 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) {
```

```
55          super(board, tile, x, y);
56          this.board = board;
57
58          this.reachable = board.getReachableMarkSystem().newMark();
59          this.mark = board.getMarkSystem().newMark();
60      }
61
62      public MutableTileInfo(MutableBoard board, TileInfo other) {
63          super(board, other);
64          this.board = board;
65
66          this.reachable = board.getReachableMarkSystem().newMark();
67          this.mark = board.getMarkSystem().newMark();
68      }
69
70      // GETTERS //
71
72      @Override
73      public boolean isDeadTile() {
74          return deadTile;
75      }
76
77      @Override
78      public boolean isReachable() {
79          return !tile.isSolid() && board.getCorral(this).containsPlayer();
80      }
81
82       @Override
83      public Tunnel getTunnel() {
84          return tunnel;
85      }
86
87      @Override
88      public Tunnel.Exit getTunnelExit() {
89          return tunnelExit;
90      }
91
92      public boolean isInATunnel() {
93          return tunnel != null;
94      }
95
96      @Override
97      public Room getRoom() {
98          return room;
99      }
100
101     @Override
102     public boolean isInARoom() {
103         return room != null;
104     }
105
106     @Override
107     public boolean isMarked() {
108         return mark.isMarked();
109     }
110
111     @Override
112     public TargetRemoteness getNearestTarget() {
113         return nearestTarget;
114     }
115
116     @Override
117     public TargetRemoteness[] getTargets() {
```

```
118         return targets;
119     }
120
121
122     // SETTERS //
123
124     @Override
125     public void addCrate() {
126         if (tile == Tile.FLOOR) {
127             tile = Tile.CRATE;
128         } else if (tile == Tile.TARGET) {
129             tile = Tile.CRATE_ON_TARGET;
130         }
131     }
132
133     @Override
134     public void removeCrate() {
135         if (tile == Tile.CRATE) {
136             tile = Tile.FLOOR;
137         } else if (tile == Tile.CRATE_ON_TARGET) {
138             tile = Tile.TARGET;
139         }
140     }
141
142     @Override
143     public void setTile(Tile tile) {
144         this.tile = tile;
145     }
146
147     @Override
148     public void setDeadTile(boolean deadTile) {
149         this.deadTile = deadTile;
150     }
151
152     @Override
153     public void setReachable(boolean reachable) {
154         this.reachable.setMarked(reachable);
155     }
156
157     @Override
158     public void setTunnel(Tunnel tunnel) {
159         this.tunnel = tunnel;
160     }
161
162     @Override
163     public void setTunnelExit(Tunnel.Exit tunnelExit) {
164         this.tunnelExit = tunnelExit;
165     }
166
167     @Override
168     public void setRoom(Room room) {
169         this.room = room;
170     }
171
172     @Override
173     public void mark() {
174         mark.mark();
175     }
176
177     @Override
178     public void unmark() {
179         mark.unmark();
180     }
```

```
181
182     @Override
183     public void setMarked(boolean marked) {
184         mark.setMarked(marked);
185     }
186
187     @Override
188     public void setTargets(TargetRemoteness[] targets) {
189         this.targets = targets;
190     }
191
192     @Override
193     public void setNearestTarget(TargetRemoteness nearestTarget) {
194         this.nearestTarget = nearestTarget;
195     }
196
197     @Override
198     public int getCrateIndex() {
199         return crateIndex;
200     }
201
202     @Override
203     public void setCrateIndex(int crateIndex) {
204         this.crateIndex = crateIndex;
205     }
206 }
```

GenericTileInfo

```
1  package fr.valax.sokoshell.solver.board.tiles;
2
3  import fr.valax.sokoshell.solver.board.Board;
4  import fr.valax.sokoshell.solver.board.Room;
5  import fr.valax.sokoshell.solver.board.Tunnel;
6
7  /**
8   * A {@code package-private} class meant to be use as a base class for {@link TileInfo}
   ↪  implementations.
9   * It defines all the basic properties and their corresponding getters
10  * (position, tile, board, etc.)
11  *
12  * @see TileInfo
13  */
14 public abstract class GenericTileInfo implements TileInfo {
15
16     protected final Board board;
17
18     protected final int x;
19     protected final int y;
20
21     protected Tile tile;
22
23
24     /**
25      * Create a new TileInfo
26      *
27      * @param tile the tile
28      * @param x the position on the x-axis in the board
29      * @param y the position on the y-axis in the board
30      */
31     public GenericTileInfo(Board board, Tile tile, int x, int y) {
32         this.board = board;
```

```
33          this.tile = tile;
34          this.x = x;
35          this.y = y;
36      }
37
38      public GenericTileInfo(TileInfo tileInfo) {
39          this(tileInfo.getBoard(), tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
40      }
41
42      public GenericTileInfo(Board board, TileInfo tileInfo) {
43          this(board, tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
44      }
45
46      @Override
47      public Tile getTile() {
48          return tile;
49      }
50
51      @Override
52      public int getX() {
53          return x;
54      }
55
56      @Override
57      public int getY() {
58          return y;
59      }
60
61      /**
62       * Returns the board in which this tile is
63       *
64       * @return the board in which this tile is
65       */
66      public Board getBoard() {
67          return board;
68      }
69
70      // SETTERS: throw UnsupportedOperationException as this class is immutable //
71
72      @Override
73      public void addCrate() {
74          throw new UnsupportedOperationException("Immutable object");
75      }
76
77      @Override
78      public void removeCrate() {
79          throw new UnsupportedOperationException("Immutable object");
80      }
81
82      @Override
83      public void setTile(Tile tile) {
84          throw new UnsupportedOperationException("Immutable object");
85      }
86
87      @Override
88      public void setDeadTile(boolean deadTile) {
89          throw new UnsupportedOperationException("Immutable object");
90      }
91
92      @Override
93      public void setReachable(boolean reachable) {
94          throw new UnsupportedOperationException("Immutable object");
95      }
```

```
96
97      @Override
98      public void setTunnel(Tunnel tunnel) {
99          throw new UnsupportedOperationException("Immutable object");
100     }
101
102     @Override
103     public void setTunnelExit(Tunnel.Exit tunnelExit) {
104         throw new UnsupportedOperationException("Immutable object");
105     }
106
107     @Override
108     public void setRoom(Room room) {
109         throw new UnsupportedOperationException("Immutable object");
110     }
111
112     @Override
113     public void mark() {
114         throw new UnsupportedOperationException("Immutable object");
115     }
116
117     @Override
118     public void unmark() {
119         throw new UnsupportedOperationException("Immutable object");
120     }
121
122     @Override
123     public void setMarked(boolean marked) {
124         throw new UnsupportedOperationException("Immutable object");
125     }
126
127     @Override
128     public void setTargets(TargetRemoteness[] targets) {
129         throw new UnsupportedOperationException("Immutable object");
130     }
131
132     @Override
133     public void setNearestTarget(TargetRemoteness nearestTarget) {
134         throw new UnsupportedOperationException("Immutable object");
135     }
136
137     @Override
138     public void setCrateIndex(int index) {
139         throw new UnsupportedOperationException("Immutable object");
140     }
141
142     @Override
143     public int hashCode() {
144         return y * board.getWidth() + x;
145     }
146 }
```

ImmutableTileInfo

```
1  package fr.valax.sokoshell.solver.board.tiles;
2
3  import fr.valax.sokoshell.solver.board.ImmutableBoard;
4  import fr.valax.sokoshell.solver.board.Room;
5  import fr.valax.sokoshell.solver.board.Tunnel;
6
7  /**
8   * 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;
    }

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

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

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

```
70
71      @Override
72      public String toString() {
73          return tile.toString();
74      }
75
76      @Override
77      public TargetRemoteness getNearestTarget() {
78          return null;
79      }
80
81      @Override
82      public TargetRemoteness[] getTargets() {
83          return null;
84      }
85
86      @Override
87      public int getCrateIndex() {
88          return -1;
89      }
90  }
```

TileInfo

```
1  package fr.valax.sokoshell.solver.board.tiles;
2
3  import fr.valax.sokoshell.solver.Corral;
4  import fr.valax.sokoshell.solver.board.*;
5  import fr.valax.sokoshell.solver.board.mark.Mark;
6  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
7
8  import java.util.List;
9
10 /**
11  * The {@link TileInfo} interface defines the methods that {@link Board} implementations need to
    ↪  manage tiles,
12  * for instance:
13  * <ul>
14  *     <li>the position</li>
15  *     <li>the {@link Tile}</li>
16  * </ul>
17  * It defines a set of high-level interactions functions.
18  *
19  * @see Board
20  */
21 public interface TileInfo {
22
23      // GETTERS //
24
25      /**
26       * @return the position of this TileInfo on the x-axis
27       */
28      int getX();
29
30      /**
31       * @return the position of this TileInfo on the y-axis
32       */
33      int getY();
34
35      /**
36       * @return which tile is this TileInfo
37       */
```

```
38      Tile getTile();
39
40      /**
41       * @return true if there is a crate at this position
42       */
43      default boolean anyCrate() {
44          return getTile().isCrate();
45      }
46
47      /**
48       * @return true if there is a wall or a crate at this position
49       */
50      default boolean isSolid() {
51          return getTile().isSolid();
52      }
53
54      /**
55       * @return true if this TileInfo is exactly a floor
56       */
57      default boolean isFloor() {
58          return getTile() == Tile.FLOOR;
59      }
60
61      /**
62       * @return true if this TileInfo is exactly a wall
63       */
64      default boolean isWall() {
65          return getTile() == Tile.WALL;
66      }
67
68      /**
69       * @return true if this TileInfo is exactly a target
70       */
71      default boolean isTarget() {
72          return getTile() == Tile.TARGET;
73      }
74
75      /**
76       * @return true if this TileInfo is exactly a crate
77       * @see #anyCrate()
78       */
79      default boolean isCrate() {
80          return getTile() == Tile.CRATE;
81      }
82
83      /**
84       * @return true if this TileInfo is exactly a crate on target
85       * @see #anyCrate()
86       */
87      default boolean isCrateOnTarget() {
88          return getTile() == Tile.CRATE_ON_TARGET;
89      }
90
91      /**
92       * Returns {@code true} if this tile is at the same position as 'other'
93       * @param other other tile
94       * @return {@code true} if this tile is at the same position as 'other'
95       */
96      default boolean isAt(TileInfo other) {
97          return isAt(other.getX(), other.getY());
98      }
99
100     /**
```

```
     * 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();
```

```
164
165     /**
166      * Returns the room in which this tile is
167      *
168      * @return the room in which this tile is
169      */
170     Room getRoom();
171
172     /**
173      * Returns {@code true} if this tile info is in a room
174      *
175      * @return {@code true} if this tile info is in a room
176      */
177     boolean isInARoom();
178
179     /**
180      * @return {@code true} if this tile is marked
181      * @see Mark
182      * @see MarkSystem
183      */
184     boolean isMarked();
185
186     /**
187      * @param dir the direction
188      * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
189      * @throws IndexOutOfBoundsException if this TileInfo is near the border of the board and
190      * the direction point outside the board
191      */
192     default TileInfo adjacent(Direction dir) {
193         return getBoard().getAt(getX() + dir.dirX(), getY() + dir.dirY());
194     }
195
196     /**
197      * @param dir the direction
198      * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
199      * or {@code null} if the adjacent tile is outside the board
200      */
201     default TileInfo safeAdjacent(Direction dir) {
202         return getBoard().safeGetAt(getX() + dir.dirX(), getY() + dir.dirY());
203     }
204
205     /**
206      * Returns the board in which this tile is
207      *
208      * @return the board in which this tile is
209      */
210     Board getBoard();
211
212     default int getIndex() {
213         return getY() * getBoard().getWidth() + getX();
214     }
215
216     /**
217      * Represents the index of this crate in {@link fr.valax.sokoshell.solver.State#cratesIndices()}
218      * array.
219      * @return -1 if not set or the index of this crate in
220      *         {@link fr.valax.sokoshell.solver.State#cratesIndices()} array.
221      */
222     int getCrateIndex();
223
224     TargetRemoteness getNearestTarget();
225
226     TargetRemoteness[] getTargets();
```

```
227
228     /**
229      * @implNote If you replace index by TileInfo, you will need to modify MutableBoard#StaticTile.
230      * If you are too lazy to do that, create an issue on github
231      */
232     record TargetRemoteness(int index, int distance) implements Comparable<TargetRemoteness> {
233
234         @Override
235         public int compareTo(TargetRemoteness other) {
236             return this.distance - other.distance;
237         }
238
239         @Override
240         public String toString() {
241             return "TR[d=" + distance + ", i=" + index + "]";
242         }
243     }
244
245
246     // SETTERS //
247
248     /**
249      * If this was a floor, this is now a crate
250      * If this was a target, this is now a crate on target
251      * @throws UnsupportedOperationException if the {@code addCrate} operation isn't
252      * supported by this TileInfo
253      */
254     void addCrate();
255
256     /**
257      * If this was a crate, this is now a floor
258      * If this was a crate on target, this is now a target
259      * @throws UnsupportedOperationException if the {@code removeCrate} operation isn't
260      * supported by this TileInfo
261      */
262     void removeCrate();
263
264     /**
265      * Sets the tile.
266      * @param tile the new tile
267      * @throws UnsupportedOperationException if the {@code setTile} operation isn't
268      * supported by this TileInfo
269      */
270     void setTile(Tile tile);
271
272     /**
273      * Sets this tile as a dead tile or not
274      * @throws UnsupportedOperationException if the {@code setDeadTile} operation isn't
275      * supported by this TileInfo
276      * @see MutableBoard#computeDeadTiles()
277      */
278     void setDeadTile(boolean deadTile);
279
280     /**
281      * Sets this tile as reachable or not by the player. It doesn't check if it's possible.
282      * @throws UnsupportedOperationException if the {@code setReachable} operation isn't
283      * supported by this TileInfo
284      * @see MutableBoard#findReachableCases(int)
285      */
286     void setReachable(boolean reachable);
287
288     /**
289      * Sets the tunnel in which this tile is
```

```
290      * @throws UnsupportedOperationException if the {@code setTunnel} operation isn't
291      * supported by this TileInfo
292      */
293     void setTunnel(Tunnel tunnel);
294 
295     /**
296      * Sets the {@link Tunnel.Exit} object associated with this tile info
297      * @throws UnsupportedOperationException if the {@code setTunnelExit} operation isn't
298      * supported by this TileInfo
299      * @see Tunnel.Exit
300      */
301     void setTunnelExit(Tunnel.Exit tunnelExit);
302 
303     /**
304      * Sets the room in which this tile is
305      * @throws UnsupportedOperationException if the {@code setRoom} operation isn't
306      * supported by this TileInfo
307      */
308     void setRoom(Room room);
309 
310     /**
311      * Sets this tile as marked
312      * @throws UnsupportedOperationException if the {@code mark} operation isn't
313      * supported by this TileInfo
314      * @see Mark
315      * @see MarkSystem
316      */
317     void mark();
318 
319     /**
320      * Sets this tile as unmarked
321      * @throws UnsupportedOperationException if the {@code unmark} operation isn't
322      * supported by this TileInfo
323      * @see Mark
324      * @see MarkSystem
325      */
326     void unmark();
327 
328     /**
329      * Sets this tile as marked or not
330      * @throws UnsupportedOperationException if the {@code setMarked} operation isn't
331      * supported by this TileInfo
332      * @see Mark
333      * @see MarkSystem
334      */
335     void setMarked(boolean marked);
336 
337     /**
338      * Set the distance to every targets
339      * @param targets distance to every targets
340      * @throws UnsupportedOperationException if the {@code setTargets} operation isn't
341      * supported by this TileInfo
342      */
343     void setTargets(TargetRemoteness[] targets);
344 
345     /**
346      * Set the nearest target
347      * @param nearestTarget nearest target
348      * @throws UnsupportedOperationException if the {@code setNearestTarget} operation isn't
349      * supported by this TileInfo
350      */
351     void setNearestTarget(TargetRemoteness nearestTarget);
352 
```

```
353     /**
354      * @see #getCrateIndex()
355      */
356     void setCrateIndex(int index);
357 }
```

Tile

```
1  package fr.valax.sokoshell.solver.board.tiles;
2
3  /**
4   * Represents the content of a case of the board.
5   */
6  public enum Tile {
7
8      FLOOR(false, false),
9      WALL(true, false),
10     CRATE(true, true),
11     CRATE_ON_TARGET(true, true),
12     TARGET(false, false);
13
14
15     private final boolean solid;
16
17     private final boolean crate;
18
19     Tile(boolean solid, boolean crate) {
20         this.solid = solid;
21         this.crate = crate;
22     }
23
24     /**
25      * Tells whether objects (i.e. player or crates) can move through the case or not.
26      */
27     public boolean isSolid() {
28         return solid;
29     }
30
31     /**
32      * Tells whether the case is occupied by a crate (on a target or not) or not.
33      */
34     public boolean isCrate() {
35         return crate;
36     }
37 }
```

1.4.2 mark

HeavyweightMarkSystem

```
1  package fr.valax.sokoshell.solver.board.mark;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * A heavyweight mark system contains a pointer to every mark associated with this system
8   */
9  public class HeavyweightMarkSystem extends AbstractMarkSystem {
10
11     protected final List<Mark> marks;
```

```
12
13      public HeavyweightMarkSystem() {
14          marks = new ArrayList<>();
15      }
16
17      @Override
18      public Mark newMark() {
19          Mark m = super.newMark();
20          marks.add(m);
21
22          return m;
23      }
24
25      @Override
26      public void reset() {
27          mark = 0;
28
29          for (Mark m : marks) {
30              m.unmark();
31          }
32      }
33  }
```

FixedSizeMarkSystem

```
1   package fr.valax.sokoshell.solver.board.mark;
2
3   public class FixedSizeMarkSystem implements MarkSystem {
4
5       protected final FMark[] marks;
6       protected int mark;
7
8       public FixedSizeMarkSystem(int capacity) {
9           marks = new FMark[capacity];
10          for (int i = 0; i < capacity; i++) {
11              marks[i] = new FMark();
12          }
13      }
14
15      public void mark(int i) {
16          marks[i].mark();
17      }
18
19      public boolean isMarked(int i) {
20          return marks[i].isMarked();
21      }
22
23      @Override
24      public Mark newMark() {
25          throw new UnsupportedOperationException();
26      }
27
28      @Override
29      public void unmarkAll() {
30          mark++;
31
32          if (mark == 0) {
33              reset();
34          }
35      }
36
37      @Override
```

```
38      public void reset() {
39          mark = 0;
40
41          for (FMark mark : marks) {
42              mark.unmark();
43          }
44      }
45
46      @Override
47      public int getMark() {
48          return mark;
49      }
50
51      private class FMark implements Mark {
52
53          private int mark = 0;
54
55          @Override
56          public void mark() {
57              mark = FixedSizeMarkSystem.this.mark;
58          }
59
60          @Override
61          public void unmark() {
62              mark = FixedSizeMarkSystem.this.mark - 1;
63          }
64
65          @Override
66          public boolean isMarked() {
67              return mark == FixedSizeMarkSystem.this.mark;
68          }
69
70          @Override
71          public MarkSystem getMarkSystem() {
72              return FixedSizeMarkSystem.this;
73          }
74      }
75  }
```

Mark

```
1   package fr.valax.sokoshell.solver.board.mark;
2
3   /**
4    * @see MarkSystem
5    * @author PoulpoGaz
6    */
7   public interface Mark {
8
9       /**
10       * Marks the object. After this method is called, {@link #isMarked()}
11       * will return {@code true}
12       */
13      void mark();
14
15      /**
16       * Un-marks the object. After this method is called, {@link #isMarked()}
17       * will return {@code false}
18       */
19      void unmark();
20
21      /**
```

```
     * 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();
    }

    @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

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

```

```java
/**
 * <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++;
```

```
21
22          if (mark == 0) {
23              reset();
24          }
25      }
26
27      @Override
28      public abstract void reset();
29
30      @Override
31      public int getMark() {
32          return mark;
33      }
34  }
```

Tunnel

```
1   package fr.valax.sokoshell.solver.board;
2
3   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
4
5   import java.util.ArrayList;
6   import java.util.List;
7
8   /**
9    * A tunnel is a zone of the board like this:
10   *
11   * <pre>
12   *      $$$$$$
13   *           $$$$$
14   *      $$$$
15   *         $$$$$$$
16   * </pre>
17   */
18  public class Tunnel {
19
20      // STATIC
21
22      protected TileInfo start;
23      protected TileInfo end;
24
25      // the tile outside the tunnel adjacent to start
26      protected TileInfo startOut;
27
28      // the tile outside the tunnel adjacent to end
29      protected TileInfo endOut;
30      protected List<Room> rooms;
31
32      // true if the tunnel can only be taken by the player
33      protected boolean playerOnlyTunnel;
34      protected boolean isOneway;
35
36
37      // DYNAMIC
38      protected boolean crateInside = false;
39
40
41
42      public void createTunnelExits() {
43          if (this.startOut != null) {
44              Direction initDir = start.direction(startOut);
45              create(start, initDir, startOut);
```

```
46          }
47
48          if (endOut != null) {
49              Direction endDir = end.direction(endOut);
50              create(end, endDir, endOut);
51          }
52      }
53
54      private void create(TileInfo tile, Direction startDir, TileInfo startOut) {
55          TileInfo t = tile;
56
57          Direction nextDir = startDir.negate();
58          while (true) {
59              TileInfo next = t.adjacent(nextDir);
60
61              if (next.isWall() || t.getTunnel() != this) {
62                  break;
63              }
64
65              setExit(t, startDir, startOut);
66
67              t = next;
68          }
69      }
70
71      private void setExit(TileInfo tile, Direction dir, TileInfo out) {
72          if (dir != null) {
73              Exit exit = tile.getTunnelExit();
74
75              if (exit == null) {
76                  exit = new Exit();
77                  tile.setTunnelExit(exit);
78              }
79
80              switch (dir) {
81                  case RIGHT -> exit.setRightExit(out);
82                  case UP -> exit.setUpExit(out);
83                  case DOWN -> exit.setDownExit(out);
84                  case LEFT -> exit.setLeftExit(out);
85              }
86          }
87      }
88
89      public void addRoom(Room room) {
90          if (rooms == null) {
91              rooms = new ArrayList<>();
92          }
93          rooms.add(room);
94      }
95
96      public List<Room> getRooms() {
97          return rooms;
98      }
99
100     public TileInfo getStart() {
101         return start;
102     }
103
104     public void setStart(TileInfo start) {
105         this.start = start;
106     }
107
108     public TileInfo getEnd() {
```

```
109         return end;
110     }
111 
112     public void setEnd(TileInfo end) {
113         this.end = end;
114     }
115 
116     public TileInfo getStartOut() {
117         return startOut;
118     }
119 
120     public void setStartOut(TileInfo startOut) {
121         this.startOut = startOut;
122     }
123 
124     public TileInfo getEndOut() {
125         return endOut;
126     }
127 
128     public void setEndOut(TileInfo endOut) {
129         this.endOut = endOut;
130     }
131 
132     public boolean isPlayerOnlyTunnel() {
133         return playerOnlyTunnel;
134     }
135 
136     public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
137         this.playerOnlyTunnel = playerOnlyTunnel;
138     }
139 
140     public boolean crateInside() {
141         return crateInside;
142     }
143 
144     public void setCrateInside(boolean crateInside) {
145         this.crateInside = crateInside;
146     }
147 
148     public boolean isOneway() {
149         return isOneway;
150     }
151 
152     public void setOneway(boolean oneway) {
153         isOneway = oneway;
154     }
155 
156     @Override
157     public String toString() {
158         if (startOut == null) {
159             return "closed - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
160                     .formatted(start.getX(), start.getY(),
161                             end.getX(), end.getY(),
162                             endOut.getX(), endOut.getY(),
163                             playerOnlyTunnel, isOneway);
164         } else if (endOut == null) {
165             return "(%d; %d) - (%d; %d) --> (%d; %d) - closed. only player? %s. one way? %s"
166                     .formatted(startOut.getX(), startOut.getY(),
167                             start.getX(), start.getY(),
168                             end.getX(), end.getY(),
169                             playerOnlyTunnel, isOneway);
170         } else {
171             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() {
        }

        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;
        }
```

```
235
236         private void setRightExit(TileInfo rightExit) {
237             this.rightExit = rightExit;
238         }
239
240         public TileInfo getDownExit() {
241             return downExit;
242         }
243
244         private void setDownExit(TileInfo downExit) {
245             this.downExit = downExit;
246         }
247     }
248 }
```

ImmutableBoard

```
1  package fr.valax.sokoshell.solver.board;
2
3  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
4  import fr.valax.sokoshell.solver.board.tiles.ImmutableTileInfo;
5  import fr.valax.sokoshell.solver.board.tiles.Tile;
6  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
7
8  import java.util.List;
9
10 /**
11  * Immutable implementation of {@link Board}.
12  *
13  * This class extends {@link GenericBoard}. It internally uses {@link ImmutableTileInfo} to store the
    ↪  board content
14  * in {@link GenericBoard#content}. As it is immutable, it implements the setters methods always
    ↪  throws a
15  * {@link UnsupportedOperationException} when such a method is called.
16  *
17  * @see Board
18  * @see GenericBoard
19  * @see TileInfo
20  */
21 public class ImmutableBoard extends GenericBoard {
22
23     public ImmutableBoard(Tile[][] content, int width, int height) {
24         super(width, height);
25
26         this.content = new ImmutableTileInfo[height][width];
27
28         for (int y = 0; y < height; y++) {
29             for (int x = 0; x < width; x++) {
30                 this.content[y][x] = new ImmutableTileInfo(this, content[y][x], x, y);
31             }
32         }
33     }
34
35     public ImmutableBoard(Board other) {
36         super(other.getWidth(), other.getHeight());
37
38         this.content = new ImmutableTileInfo[height][width];
39
40         for (int y = 0; y < height; y++) {
41             for (int x = 0; x < width; x++) {
42                 this.content[y][x] = new ImmutableTileInfo(other.getAt(x, y));
43             }
```

```
        }
    }

    // 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;
```

```
24
25      Move(String name, Direction direction, boolean moveCrate) {
26          this.shortName = name;
27          this.direction = direction;
28          this.moveCrate = moveCrate;
29      }
30
31      public String shortName() {
32          return shortName;
33      }
34
35      public Direction direction() {
36          return direction;
37      }
38
39      public boolean moveCrate() {
40          return moveCrate;
41      }
42
43      public static Move of(Direction dir, boolean moveCrate) {
44          return switch (dir) {
45              case LEFT -> moveCrate ? LEFT_PUSH : LEFT;
46              case UP -> moveCrate ? UP_PUSH : UP;
47              case DOWN -> moveCrate ? DOWN_PUSH : DOWN;
48              case RIGHT -> moveCrate ? RIGHT_PUSH : RIGHT;
49          };
50      }
51
52      public static Move of(String shortName) {
53          for (Move move : Move.values()) {
54              if (move.shortName().equals(shortName)) {
55                  return move;
56              }
57          }
58
59          return null;
60      }
61  }
```

Room

```
1  package fr.valax.sokoshell.solver.board;
2
3  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
4
5  import java.util.ArrayList;
6  import java.util.List;
7
8  public class Room {
9
10     protected boolean goalRoom;
11
12     protected final List<TileInfo> tiles = new ArrayList<>();
13     protected final List<TileInfo> targets = new ArrayList<>();
14
15     protected List<Tunnel> tunnels;
16
17     /**
18      * Only computed if the level is a goal room level as defined by {@link Board#isGoalRoomLevel()}
19      */
20     protected List<TileInfo> packingOrder;
21
```

```
22      // dynamic
23      // the index in packingOrder of the position of the next crate that will be pushed inside the room
24      // negative if it is not possible because a crate isn't at the correct position
25      // or if the room isn't a goal room
26      protected int packingOrderIndex;
27
28      public Room() {
29      }
30
31      public void addTile(TileInfo tile) {
32          tiles.add(tile);
33
34          if (tile.isTarget()) {
35              targets.add(tile);
36          }
37      }
38
39
40      public List<TileInfo> getTiles() {
41          return tiles;
42      }
43
44      public List<TileInfo> getTargets() {
45          return targets;
46      }
47
48
49      public void addTunnel(Tunnel tunnel) {
50          if (tunnels == null) {
51              tunnels = new ArrayList<>();
52          }
53          tunnels.add(tunnel);
54      }
55
56      public List<Tunnel> getTunnels() {
57          return tunnels;
58      }
59
60
61      public boolean isGoalRoom() {
62          return goalRoom;
63      }
64
65      public void setGoalRoom(boolean goalRoom) {
66          this.goalRoom = goalRoom;
67      }
68
69      public List<TileInfo> getPackingOrder() {
70          return packingOrder;
71      }
72
73      public void setPackingOrder(List<TileInfo> packingOrder) {
74          this.packingOrder = packingOrder;
75      }
76
77      public boolean isInPackingOrder(TileInfo tile) {
78          return packingOrder != null && packingOrder.contains(tile);
79      }
80
81      public int getPackingOrderIndex() {
82          return packingOrderIndex;
83      }
84
```

```
85      public void setPackingOrderIndex(int packingOrderIndex) {
86          this.packingOrderIndex = packingOrderIndex;
87      }
88  }
```

MutableBoard

```
1   package fr.valax.sokoshell.solver.board;
2
3   import fr.valax.sokoshell.SokoShell;
4   import fr.valax.sokoshell.graphics.Surface;
5   import fr.valax.sokoshell.solver.Corral;
6   import fr.valax.sokoshell.solver.CorralDetector;
7   import fr.valax.sokoshell.solver.State;
8   import fr.valax.sokoshell.solver.board.mark.AbstractMarkSystem;
9   import fr.valax.sokoshell.solver.board.mark.Mark;
10  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
11  import fr.valax.sokoshell.solver.board.tiles.GenericTileInfo;
12  import fr.valax.sokoshell.solver.board.tiles.MutableTileInfo;
13  import fr.valax.sokoshell.solver.board.tiles.Tile;
14  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
15  import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
16  import fr.valax.sokoshell.solver.pathfinder.CratePlayerAStar;
17  import fr.valax.sokoshell.solver.pathfinder.PlayerAStar;
18
19  import java.util.*;
20  import java.util.function.Consumer;
21
22
23  /**
24   * Mutable implementation of {@link Board}.
25   *
26   * This class extends {@link GenericBoard} by defining all the setters methods. It internally uses
    ↪  {@link MutableTileInfo} to store the board content
27   * in {@link GenericBoard#content}.
28   *
29   * @see Board
30   * @see GenericBoard
31   * @see MutableTileInfo
32   */
33  @SuppressWarnings("ForLoopReplaceableByForEach")
34  public class MutableBoard extends GenericBoard {
35
36      private final MarkSystem markSystem = newMarkSystem(TileInfo::unmark);
37      private final MarkSystem reachableMarkSystem = newMarkSystem((t) -> t.setReachable(false));
38
39      private int targetCount;
40
41      /**
42       * Tiles that can be 'target' or 'floor'
43       */
44      private TileInfo[] floors;
45
46      private final List<Tunnel> tunnels = new ArrayList<>();
47      private final List<Room> rooms = new ArrayList<>();
48
49
50      /**
51       * True if all rooms are goal room with only one entrance
52       */
53      private boolean isGoalRoomLevel;
54
```

```
55      private PlayerAStar playerAStar;
56      private CrateAStar crateAStar;
57      private CratePlayerAStar cratePlayerAStar;
58
59      private final CorralDetector corralDetector;
60
61      private StaticBoard staticBoard;
62
63      /**
64       * Creates a SolverBoard with the specified width, height and tiles
65       *
66       * @param content a rectangular matrix of size width * height. The first index is for the rows
67       *                and the second for the columns
68       * @param width board width
69       * @param height board height
70       */
71      public MutableBoard(Tile[][] content, int width, int height) {
72          super(width, height);
73
74          this.content = new TileInfo[height][width];
75          for (int y = 0; y < height; y++) {
76              for (int x = 0; x < width; x++) {
77                  this.content[y][x] = new MutableTileInfo(this, content[y][x], x, y);
78              }
79          }
80
81          corralDetector = new CorralDetector(this);
82      }
83
84      public MutableBoard(int width, int height) {
85          super(width, height);
86
87          this.content = new TileInfo[height][width];
88          for (int y = 0; y < height; y++) {
89              for (int x = 0; x < width; x++) {
90                  this.content[y][x] = new MutableTileInfo(this, Tile.FLOOR, x, y);
91              }
92          }
93
94          corralDetector = new CorralDetector(this);
95      }
96
97      /**
98       * Creates a copy of 'other'. It doesn't copy solver information
99       *
100      * @param other the board to copy
101      */
102     public MutableBoard(Board other) {
103         this(other, false);
104     }
105
106     public MutableBoard(Board other, boolean copyStatic) {
107         super(other.getWidth(), other.getHeight());
108
109         content = new TileInfo[height][width];
110         for (int y = 0; y < height; y++) {
111             for (int x = 0; x < width; x++) {
112                 content[y][x] = new MutableTileInfo(this, other.getAt(x, y));
113             }
114         }
115
116         corralDetector = new CorralDetector(this);
117
```

```
118         if (copyStatic) {
119             copyStaticInformation(other);
120         }
121     }
122 
123     private void copyStaticInformation(Board other) {
124         // map room in other board and in this board
125         Map<Room, Room> roomMap = new HashMap<>(rooms.size());
126         Map<Tunnel, Tunnel> tunnelMap = new HashMap<>(rooms.size());
127 
128         // copy tunnels, rooms
129         for (Room room : other.getRooms()) {
130             Room copy = copyRoom(room);
131             roomMap.put(room, copy);
132             rooms.add(copy);
133         }
134         for (Tunnel tunnel : other.getTunnels()) {
135             Tunnel copy = copyTunnel(tunnel);
136             tunnelMap.put(tunnel, copy);
137             tunnels.add(copy);
138         }
139 
140         // copy tile info
141         for (int y = 0; y < height; y++) {
142             for (int x = 0; x < width; x++) {
143                 TileInfo otherTile = other.getAt(x, y);
144                 TileInfo tile = content[y][x];
145                 tile.setDeadTile(otherTile.isDeadTile());
146 
147                 if (tile.getTargets() != null) {
148                     tile.setTargets(Arrays.copyOf(tile.getTargets(), tile.getTargets().length));
149                 }
150                 tile.setNearestTarget(otherTile.getNearestTarget());
151 
152                 tile.setTunnel(tunnelMap.get(otherTile.getTunnel()));
153                 tile.setRoom(roomMap.get(otherTile.getRoom()));
154                 if (otherTile.getTunnelExit() != null) {
155                     tile.setTunnelExit(otherTile.getTunnelExit()); // it is immutable !
156                 }
157             }
158         }
159 
160         // link rooms and tunnels
161         for (Tunnel tunnel : other.getTunnels()) {
162             Tunnel newTunnel = tunnelMap.get(tunnel);
163             for (Room room : other.getRooms()) {
164                 Room newRoom = roomMap.get(room);
165                 newTunnel.addRoom(newRoom);
166                 newRoom.addTunnel(newTunnel);
167             }
168         }
169     }
170 
171     private Room copyRoom(Room room) {
172         Room newRoom = new Room();
173         newRoom.setGoalRoom(room.isGoalRoom());
174 
175         for (TileInfo t : room.getTiles()) {
176             newRoom.addTile(getAt(t.getIndex()));
177         }
178         if (room.getPackingOrder() != null) {
179             List<TileInfo> packingOrder = new ArrayList<>();
180             for (TileInfo t : room.getPackingOrder()) {
```

```
181                 packingOrder.add(getAt(t.getIndex()));
182             }
183             newRoom.setPackingOrder(packingOrder);
184         }
185 
186         return newRoom;
187     }
188 
189     private Tunnel copyTunnel(Tunnel tunnel) {
190         Tunnel newTunnel = new Tunnel();
191 
192         newTunnel.setStart(getAt(tunnel.getStart().getIndex()));
193         newTunnel.setEnd(getAt(tunnel.getEnd().getIndex()));
194 
195         if (tunnel.getStartOut() != null) {
196             newTunnel.setStartOut(getAt(tunnel.getStartOut().getIndex()));
197         }
198         if (tunnel.getEndOut() != null) {
199             newTunnel.setEndOut(getAt(tunnel.getEndOut().getIndex()));
200         }
201         newTunnel.setPlayerOnlyTunnel(tunnel.isPlayerOnlyTunnel());
202         newTunnel.setOneway(tunnel.isOneway());
203 
204         return newTunnel;
205     }
206 
207     /**
208      * Apply the consumer on every tile info
209      *
210      * @param consumer the consumer to apply
211      */
212     public void forEach(Consumer<TileInfo> consumer) {
213         for (int y = 0; y < height; y++) {
214             for (int x = 0; x < width; x++) {
215                 consumer.accept(content[y][x]);
216             }
217         }
218     }
219 
220     /**
221      * Set at tile at the specified index. The index will be converted to
222      * cartesian coordinate with {@link #getX(int)} and {@link  #getY(int)}
223      *
224      * @param index index in the board
225      * @param tile the new tile
226      * @throws IndexOutOfBoundsException if the index lead to a position outside the board
227      */
228     public void setAt(int index, Tile tile) { content[getY(index)][getX(index)].setTile(tile); }
229 
230     /**
231      * Set at tile at (x, y)
232      *
233      * @param x x position in the board
234      * @param y y position in the board
235      * @throws IndexOutOfBoundsException if the position is outside the board
236      */
237     public void setAt(int x, int y, Tile tile) {
238         content[y][x].setTile(tile);
239     }
240 
241     /**
242      * Puts the crates of the given state in the content array.
243      *
```

```
244      * @param state The state with the crates
245      */
246     public void addStateCrates(State state) {
247         int[] cratesIndices = state.cratesIndices();
248         for (int j = 0; j < cratesIndices.length; j++) {
249             int i = cratesIndices[j];
250             TileInfo crate = getAt(i);
251             crate.setCrateIndex(j);
252             crate.addCrate();
253         }
254     }
255 
256     /**
257      * Removes the crates of the given state from the content array.
258      *
259      * @param state The state with the crates
260      */
261     public void removeStateCrates(State state) {
262         for (int i : state.cratesIndices()) {
263             TileInfo crate = getAt(i);
264             crate.setCrateIndex(-1);
265             crate.removeCrate();
266         }
267     }
268 
269     /**
270      * Puts the crates of the given state in the content array.
271      * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
272      *
273      * @param state The state with the crates
274      */
275     public void safeAddStateCrates(State state) {
276         for (int i : state.cratesIndices()) {
277             TileInfo info = safeGetAt(i);
278 
279             if (info != null) {
280                 info.addCrate();
281             }
282         }
283     }
284 
285     /**
286      * Removes the crates of the given state from the content array.
287      * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
288      *
289      * @param state The state with the crates
290      */
291     public void safeRemoveStateCrates(State state) {
292         for (int i : state.cratesIndices()) {
293             TileInfo info = safeGetAt(i);
294 
295             if (info != null) {
296                 info.removeCrate();
297             }
298         }
299     }
300 
301     // ==========================================
302     // *         Methods used by solvers         *
303     // * You need to call #initForSolver() first *
304     // ==========================================
305 
306     /**
```

```
307      * Initialize the board for solving:
308      * <ul>
309      *     <li>compute floor tiles: an array containing all non-wall tile</li>
310      *     <li>compute {@linkplain #computeDeadTiles() dead tiles}</li>
311      *     <li>find {@linkplain #findTunnels() tunnels}</li>
312      * </ul>
313      * <strong>The board must have no crate inside</strong>
314      * @see Tunnel
315      */
316     public void initForSolver() {
317         playerAStar = new PlayerAStar(this);
318         crateAStar = new CrateAStar(this);
319         cratePlayerAStar = new CratePlayerAStar(this);
320 
321         computeFloors();
322         computeDeadTiles();
323         findTunnels();
324         findRooms();
325         removeUselessTunnels();
326         finishComputingTunnels();
327         tryComputePackingOrder();
328         computeTileToTargetsDistances();
329 
330         // we must compute the static board here
331         // this is the unique point where the board
332         // information are guaranteed to be true.
333         // For example, the freeze deadlock detector
334         // places wall on the map but this object
335         // has no information about this.
336         staticBoard = new StaticBoard();
337     }
338 
339     /**
340      * Creates or recreates the floor array. It is an array containing all tile info
341      * that are not a wall
342      */
343     public void computeFloors() {
344         int nFloor = 0;
345         for (int y = 0; y < height; y++) {
346             for (int x = 0; x < width; x++) {
347                 TileInfo t = getAt(x, y);
348 
349                 if (!t.isSolid() || t.isCrate()) {
350                     nFloor++;
351                 }
352             }
353         }
354 
355         this.floors = new TileInfo[nFloor];
356         int i = 0;
357         for (int y = 0; y < height; y++) {
358             for (int x = 0; x < width; x++) {
359                 if (!this.content[y][x].isSolid() || this.content[y][x].isCrate()) {
360                     this.floors[i] = this.content[y][x];
361                     i++;
362                 }
363             }
364         }
365     }
366 
367     /**
368      * Apply the consumer on every tile info except walls
369      *
```

```
     * @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);
        }

        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, as described
     * <a
    ↪  href="www.sokobano.de/wiki/index.php?title=How_to_detect_deadlocks#Detecting_simple_deadlocks">here</a>
     */
    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;
```

```
556                 } else {
557                     return null; // too many direction
558                 }
559             }
560         }
561 
562         if (pushDir1 == null) { // all adjacents tiles are wall, ie init is alone, nerver happen see
        ↪  LevelBuilder
563             return null;
564         } else if (pushDir2 == null) {
565             /*
566                 We are in this case:
567                   |$|
568                  $| |$
569              */
570 
571             Tunnel tunnel = new Tunnel();
572             tunnel.setStart(init);
573             tunnel.setEnd(init);
574             tunnel.setEndOut(init.adjacent(pushDir1));
575             init.setTunnel(tunnel);
576 
577             growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
578             return tunnel;
579         } else {
580             /*
581                 Either:
582                 #| |#
583                 Either:
584                  |#|
585                 #| |
586              */
587             boolean onlyPlayer = false;
588 
589             if (pushDir1.negate() != pushDir2) {
590                 /*
591                     First case:
592                       |#|
593                      #|i|
594                       | |#
595                     if init is like this, then this is a tunnel and a crate
596                     mustn't be pushed inside.
597 
598                     Second case:
599                       |#|
600                      #|i|
601                       | |
602                     ie not tunnel
603                 */
604                 if (init.adjacent(pushDir1).adjacent(pushDir2).isSolid()) {
605                     onlyPlayer = true;
606                 } else {
607                     return null;
608                 }
609             }
610 
611             Tunnel tunnel = new Tunnel();
612             tunnel.setEnd(init);
613             tunnel.setEndOut(init.adjacent(pushDir1));
614             tunnel.setPlayerOnlyTunnel(onlyPlayer);
615             init.setTunnel(tunnel);
616 
617             growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
```

```
618             tunnel.setStart(tunnel.getEnd());
619             tunnel.setStartOut(tunnel.getEndOut());
620             growTunnel(tunnel, init.adjacent(pushDir2), pushDir2);
621 
622             return tunnel;
623         }
624     }
625 
626     /**
627      * Try to grow a tunnel by the end ie Tunnel#end and Tunnel#endOut are modified.
628      * The tile adjacent to pos according to -dir is assumed to
629      * be a part of a tunnel. So we are in the following situations:
630      * <pre>
631      *             $$$     $$$
632      *     $ $       $     $
633      *     $@$     $@$     $@$
634      * </pre>
635      *
636      * @param pos position of the player
637      * @param dir the move the player did to go to pos
638      */
639     private void growTunnel(Tunnel t, TileInfo pos, Direction dir) {
640         pos.mark();
641 
642         Direction leftDir = dir.left();
643         Direction rightDir = dir.right();
644         TileInfo left = pos.adjacent(leftDir);
645         TileInfo right = pos.adjacent(rightDir);
646         TileInfo front = pos.adjacent(dir);
647 
648         if (!pos.isTarget()) {
649             pos.setTunnel(t);
650             if (left.isSolid() && right.isSolid() && front.isSolid()) {
651                 t.setPlayerOnlyTunnel(true);
652                 t.setEnd(pos);
653                 t.setEndOut(null);
654                 return;
655             } else if (left.isSolid() && right.isSolid()) {
656                 if (front.isMarked()) {
657                     t.setEnd(pos);
658                     t.setEndOut(front);
659                 } else {
660                     growTunnel(t, front, dir);
661                 }
662                 return;
663             } else if (right.isSolid() && front.isSolid()) {
664                 t.setPlayerOnlyTunnel(true);
665                 if (left.isMarked()) {
666                     t.setEnd(pos);
667                     t.setEndOut(left);
668                 } else {
669                     growTunnel(t, left, leftDir);
670                 }
671                 return;
672             } else if (left.isSolid() && front.isSolid()) {
673                 t.setPlayerOnlyTunnel(true);
674                 if (right.isMarked()) {
675                     t.setEnd(pos);
676                     t.setEndOut(right);
677                 } else {
678                     growTunnel(t, right, rightDir);
679                 }
680                 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

```

```
744                 if (room.tunnels.size() == 2 && room.tiles.size() == 1 && !room.isGoalRoom()) {
745                     // room is now useless
746                     // we are in one of the following cases:
747                     // ###    # #
748                     //     or #
749                     // #_#    #_#
750                     // _ indicates the tunnel to remove
751 
752                     // dir is the direction the player need to take to exit the tunnel
753                     Direction dir;
754                     if (t.getStartOut() == null) {
755                         dir = t.getEnd().direction(t.getEndOut());
756                     } else {
757                         dir = t.getStart().direction(t.getStartOut());
758                     }
759 
760                     Tunnel t1 = room.tunnels.get(0);
761                     Tunnel t2 = room.tunnels.get(1);
762                     TileInfo roomTile = room.getTiles().get(0);
763 
764                     merge(t1, t2, room);
765                     if (!roomTile.adjacent(dir).isSolid()) {
766                         // second case
767                         // tunnel became in every case player only
768                         t1.setPlayerOnlyTunnel(true);
769                     }
770 
771                     // remove t2, taking care of i
772                     int j = tunnels.indexOf(t2);
773                     tunnels.remove(j);
774                     if (j < i) {
775                         i--;
776                     }
777                 }
778 
779                 tunnels.remove(i);
780                 forEachNotWall((tunnel) -> {
781                     if (tunnel.getTunnel() == t) {
782                         tunnel.setTunnel(null);
783                     }
784                 });
785                 i--;
786             }
787         }
788     }
789 
790     /**
791      * Merge two tunnels, t1 will hold the result.
792      * For each tunnel, start, end, startOut, endOut, playerOnlyTunnel, rooms are updated.
793      * For each tile in t2, tunnel is replaced by t1
794      */
795     private void merge(Tunnel t1, Tunnel t2, Room room) {
796         TileInfo toAdd = room.getTiles().get(0);
797 
798         if (t1.getStartOut() == toAdd) {
799             if (t2.getStartOut() == toAdd) {
800                 t1.setStart(t2.getEnd());
801                 t1.setStartOut(t2.getEndOut());
802             } else {
803                 t1.setStart(t2.getStart());
804                 t1.setStartOut(t2.getStartOut());
805             }
806         } 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);
                        }
                    }
                }
            }
        }

        if (!newFrontier.isEmpty()) {
```

```
994             findAccessibleCrates(newFrontier, lastFrontier, out);
995         } else {
996             lastFrontier.clear();
997         }
998     }
999 
1000    private void computeTileToTargetsDistances() {
1001
1002        List<Integer> targetIndices = new ArrayList<>();
1003
1004        targetCount = 0;
1005        for (int y = 0; y < height; y++) {
1006            for (int x = 0; x < width; x++) {
1007                if (this.content[y][x].isTarget() || this.content[y][x].isCrateOnTarget()) {
1008                    targetCount++;
1009                    targetIndices.add(getIndex(x, y));
1010                }
1011            }
1012        }
1013
1014        for (int y = 0; y < height; y++) {
1015            for (int x = 0; x < width; x++) {
1016
1017                final TileInfo t = getAt(x, y);
1018
1019                int minDistToTarget = Integer.MAX_VALUE;
1020                int minDistToTargetIndex = -1;
1021
1022                getAt(x, y).setTargets(new TileInfo.TargetRemoteness[targetIndices.size()]);
1023
1024                for (int j = 0; j < targetIndices.size(); j++) {
1025
1026                    final int targetIndex = targetIndices.get(j);
1027                    final int d = (t.isFloor() || t.isTarget()
1028                                   ? playerAStar.findPath(t, getAt(targetIndex), null, null).getDist()
1029                                   : 0);
1030
1031
1032                    if (d < minDistToTarget) {
1033                        minDistToTarget = d;
1034                        minDistToTargetIndex = j;
1035                    }
1036
1037                    getAt(x, y).getTargets()[j] = new TileInfo.TargetRemoteness(targetIndex, d);
1038                }
1039                Arrays.sort(getAt(x, y).getTargets());
1040                getAt(x, y).setNearestTarget(new TileInfo.TargetRemoteness(minDistToTargetIndex,
                    ↪ minDistToTarget));
1041            }
1042        }
1043    }
1044
1045
1046
1047
1048
1049    // * DYNAMIC *
1050
1051    /**
1052     * Find reachable tiles
1053     * @param playerPos The indic of the case on which the player currently is.
1054     */
1055    public void findReachableCases(int playerPos) {
```

```
1056        findReachableCases(getAt(playerPos));
1057    }
1058
1059    public void findReachableCases(TileInfo tile) {
1060        reachableMarkSystem.unmarkAll();
1061        findReachableCases_aux(tile);
1062    }
1063
1064    private void findReachableCases_aux(TileInfo tile) {
1065        tile.setReachable(true);
1066        for (Direction d : Direction.VALUES) {
1067            TileInfo adjacent = tile.adjacent(d);
1068
1069            // the second part of the condition avoids to check already processed cases
1070            if (!adjacent.isSolid() && !adjacent.isReachable()) {
1071                findReachableCases_aux(adjacent);
1072            }
1073        }
1074    }
1075
1076
1077
1078    private int topX = 0;
1079    private int topY = 0;
1080
1081    /**
1082     * This method compute the top left reachable position of the player of pushing a crate
1083     * at (crateToMoveX, crateToMoveY) to (destX, destY). It is used to calculate the position
1084     * of the player in a {@link State}.
1085     * This is also an example of use of {@link MarkSystem}
1086     *
1087     * @return the top left reachable position after pushing the crate
1088     * @see MarkSystem
1089     * @see Mark
1090     */
1091    @Override
1092    public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
1093        // temporary move the crate
1094        crate.removeCrate();
1095        crateDest.addCrate();
1096
1097        topX = width;
1098        topY = height;
1099
1100        markSystem.unmarkAll();
1101        topLeftReachablePosition_aux(crate);
1102
1103        // undo
1104        crate.addCrate();
1105        crateDest.removeCrate();
1106
1107        return topY * width + topX;
1108    }
1109
1110    private void topLeftReachablePosition_aux(TileInfo tile) {
1111        if (tile.getY() < topY || (tile.getY() == topY && tile.getX() < topX)) {
1112            topX = tile.getX();
1113            topY = tile.getY();
1114        }
1115
1116        tile.mark();
1117        for (Direction d : Direction.VALUES) {
1118            TileInfo adjacent = tile.adjacent(d);
```

```
1119
1120            if (!adjacent.isSolid() && !adjacent.isMarked()) {
1121                topLeftReachablePosition_aux(adjacent);
1122            }
1123        }
1124    }
1125
1126
1127    // *********************
1128    // * GETTERS / SETTERS *
1129    // *********************
1130
1131    public StaticBoard staticBoard() {
1132        return staticBoard;
1133    }
1134
1135    /**
1136     * Returns the number of target i.e. tiles on which a crate has to be pushed to solve the level on
    ↪  the board
1137     * @return the number of target i.e. tiles on which a crate has to be pushed to solve the level on
    ↪  the board
1138     */
1139    public int getTargetCount() {
1140        return targetCount;
1141    }
1142
1143
1144    /**
1145     * Returns all tunnels that are in this board
1146     *
1147     * @return all tunnels that are in this board
1148     */
1149    public List<Tunnel> getTunnels() {
1150        return tunnels;
1151    }
1152
1153    /**
1154     * Returns all rooms that are in this board
1155     *
1156     * @return all rooms that are in this board
1157     */
1158    public List<Room> getRooms() {
1159        return rooms;
1160    }
1161
1162    public boolean isGoalRoomLevel() {
1163        return isGoalRoomLevel;
1164    }
1165
1166    public PlayerAStar getPlayerAStar() {
1167        return playerAStar;
1168    }
1169
1170    public CrateAStar getCrateAStar() {
1171        return crateAStar;
1172    }
1173
1174    public CratePlayerAStar getCratePlayerAStar() {
1175        return cratePlayerAStar;
1176    }
1177
1178    @Override
1179    public Corral getCorral(TileInfo tile) {
```

```
1180         return corralDetector.findCorral(tile);
1181     }
1182 
1183     @Override
1184     public CorralDetector getCorralDetector() {
1185         return corralDetector;
1186     }
1187 
1188     /**
1189      * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking twice  a tile
1190      *
1191      * @return a mark system
1192      * @see MarkSystem
1193      */
1194     public MarkSystem getMarkSystem() {
1195         return markSystem;
1196     }
1197 
1198     /**
1199      * Returns the {@linkplain MarkSystem mark system} used by the {@link #findReachableCases(int)}
    ↪  algorithm
1200      *
1201      * @return the reachable mark system
1202      * @see MarkSystem
1203      */
1204     public MarkSystem getReachableMarkSystem() {
1205         return reachableMarkSystem;
1206     }
1207 
1208     /**
1209      * Creates a {@linkplain MarkSystem mark system} that apply the specified reset
1210      * consumer to every <strong>non-wall</strong> {@linkplain TileInfo tile info}
1211      * that are in this {@linkplain  Board board}.
1212      *
1213      * @param reset the reset function
1214      * @return a new MarkSystem
1215      * @see MarkSystem
1216      * @see Mark
1217      */
1218     private MarkSystem newMarkSystem(Consumer<TileInfo> reset) {
1219         return new AbstractMarkSystem() {
1220             @Override
1221             public void reset() {
1222                 mark = 0;
1223                 forEachNotWall(reset);
1224             }
1225         };
1226     }
1227 
1228     protected class StaticBoard extends GenericBoard {
1229 
1230         private final List<ImmutableTunnel> tunnels;
1231         private final List<ImmutableRoom> rooms;
1232 
1233         public StaticBoard() {
1234             super(MutableBoard.this.width, MutableBoard.this.height);
1235 
1236             StaticTile[][] content = new StaticTile[height][width];
1237             this.content = content;
1238 
1239             for (int y = 0; y < height; y++) {
1240                 for (int x = 0; x < width; x++) {
1241                     content[y][x] = new StaticTile(this, MutableBoard.this.content[y][x]);
```

```
1242                }
1243            }
1244
1245            tunnels = MutableBoard.this.tunnels.stream()
1246                    .map((t) -> new ImmutableTunnel(this, t)).toList();
1247            rooms = MutableBoard.this.rooms.stream()
1248                    .map((r) -> new ImmutableRoom(this, r)).toList();
1249
1250            linkTunnelsRoomsAndTileInfos(content);
1251        }
1252
1253        private void linkTunnelsRoomsAndTileInfos(StaticTile[][] content) {
1254            Map<Room, ImmutableRoom> roomMap = new HashMap<>(rooms.size());
1255            for (int i = 0; i < rooms.size(); i++) {
1256                roomMap.put(MutableBoard.this.rooms.get(i), rooms.get(i));
1257            }
1258
1259            Map<Tunnel, ImmutableTunnel> tunnelMap = new HashMap<>(tunnels.size());
1260            for (int i = 0; i < tunnels.size(); i++) {
1261                tunnelMap.put(MutableBoard.this.tunnels.get(i), tunnels.get(i));
1262            }
1263
1264            // add rooms to tunnels
1265            List<Tunnel> originalTunnel = MutableBoard.this.tunnels;
1266            for (int i = 0; i < tunnels.size(); i++) {
1267                ImmutableTunnel t = tunnels.get(i);
1268                if (originalTunnel.get(i).rooms != null) {
1269                    t.rooms = originalTunnel.get(i).rooms.stream()
1270                            .map(r -> (Room) roomMap.get(r)).toList();
1271                }
1272            }
1273
1274            // add tunnels to rooms
1275            List<Room> originalRooms = MutableBoard.this.rooms;
1276            for (int i = 0; i < rooms.size(); i++) {
1277                ImmutableRoom r = rooms.get(i);
1278                if (originalRooms.get(i).tunnels != null) {
1279                    r.tunnels = originalRooms.get(i).tunnels.stream()
1280                            .map(t -> (Tunnel) tunnelMap.get(t)).toList();
1281                }
1282            }
1283
1284            // add tunnels, rooms to tile info
1285            for (int y = 0; y < getHeight(); y++) {
1286                for (int x = 0; x < getWidth(); x++) {
1287                    TileInfo original = MutableBoard.this.content[y][x];
1288                    StaticTile dest = content[y][x];
1289
1290                    dest.tunnel = tunnelMap.get(original.getTunnel());
1291                    dest.room = roomMap.get(original.getRoom());
1292
1293                    if (original.getTunnelExit() != null) {
1294                        dest.exit = original.getTunnelExit(); // it is immutable !
1295                    }
1296                }
1297            }
1298        }
1299
1300        @Override
1301        public int getWidth() {
1302            return MutableBoard.this.getWidth();
1303        }
1304
```

```
1305        @Override
1306        public int getHeight() {
1307            return MutableBoard.this.getHeight();
1308        }
1309
1310        @Override
1311        public int getTargetCount() {
1312            return MutableBoard.this.getTargetCount();
1313        }
1314
1315        @SuppressWarnings("unchecked")
1316        @Override
1317        public List<Tunnel> getTunnels() {
1318            return (List<Tunnel>) ((List<?>) tunnels); // this is black magic
1319        }
1320
1321        @SuppressWarnings("unchecked")
1322        @Override
1323        public List<Room> getRooms() {
1324            return (List<Room>) ((List<?>) rooms); // more black magic !
1325        }
1326
1327        @Override
1328        public boolean isGoalRoomLevel() {
1329            return MutableBoard.this.isGoalRoomLevel();
1330        }
1331
1332        @Override
1333        public MarkSystem getMarkSystem() {
1334            return null;
1335        }
1336
1337        @Override
1338        public MarkSystem getReachableMarkSystem() {
1339            return null;
1340        }
1341    }
1342
1343    /**
1344     * A TileInfo that contains only static information
1345     */
1346    protected static class StaticTile extends GenericTileInfo {
1347
1348        private final boolean deadTile;
1349
1350        private final TargetRemoteness[] targets;
1351        private final TargetRemoteness nearestTarget;
1352
1353        private ImmutableTunnel tunnel;
1354        private ImmutableRoom room;
1355        private Tunnel.Exit exit;
1356
1357        public StaticTile(StaticBoard staticBoard, TileInfo tile) {
1358            super(staticBoard, removeCrate(tile.getTile()), tile.getX(), tile.getY());
1359            this.deadTile = tile.isDeadTile();
1360
1361            if (tile.getTargets() == null) {
1362                targets = null;
1363            } else {
1364                targets = Arrays.copyOf(tile.getTargets(), tile.getTargets().length);
1365            }
1366
1367            this.nearestTarget = tile.getNearestTarget();
```

```
1368        }
1369
1370        private static Tile removeCrate(Tile tile) {
1371            if (tile == Tile.CRATE) {
1372                return Tile.FLOOR;
1373            } else if (tile == Tile.CRATE_ON_TARGET) {
1374                return Tile.TARGET;
1375            } else {
1376                return tile;
1377            }
1378        }
1379
1380        @Override
1381        public boolean isDeadTile() {
1382            return deadTile;
1383        }
1384
1385        @Override
1386        public boolean isReachable() {
1387            return false;
1388        }
1389
1390        @Override
1391        public Tunnel getTunnel() {
1392            return tunnel;
1393        }
1394
1395        @Override
1396        public Tunnel.Exit getTunnelExit() {
1397            return exit;
1398        }
1399
1400        @Override
1401        public boolean isInATunnel() {
1402            return tunnel != null;
1403        }
1404
1405        @Override
1406        public Room getRoom() {
1407            return room;
1408        }
1409
1410        @Override
1411        public boolean isInARoom() {
1412            return room != null;
1413        }
1414
1415        @Override
1416        public boolean isMarked() {
1417            return false;
1418        }
1419
1420        @Override
1421        public int getCrateIndex() {
1422            return -1;
1423        }
1424
1425        @Override
1426        public TargetRemoteness getNearestTarget() {
1427            return nearestTarget;
1428        }
1429
1430        @Override
```

```
1431        public TargetRemoteness[] getTargets() {
1432            return targets;
1433        }
1434    }
1435
1436    private static class ImmutableTunnel extends Tunnel {
1437
1438        public ImmutableTunnel(StaticBoard board, Tunnel tunnel) {
1439            start = board.getAt(tunnel.start.getIndex());
1440            end = board.getAt(tunnel.end.getIndex());
1441
1442            if (startOut != null) {
1443                startOut = board.getAt(tunnel.startOut.getIndex());
1444            }
1445            if (endOut != null) {
1446                endOut = board.getAt(tunnel.endOut.getIndex());
1447            }
1448            playerOnlyTunnel = tunnel.isPlayerOnlyTunnel();
1449            isOneway = tunnel.isOneway();
1450        }
1451
1452        @Override
1453        public void createTunnelExits() {
1454            throw new UnsupportedOperationException();
1455        }
1456
1457        @Override
1458        public void addRoom(Room room) {
1459            throw new UnsupportedOperationException();
1460        }
1461
1462        @Override
1463        public void setStart(TileInfo start) {
1464            throw new UnsupportedOperationException();
1465        }
1466
1467        @Override
1468        public void setEnd(TileInfo end) {
1469            throw new UnsupportedOperationException();
1470        }
1471
1472        @Override
1473        public void setStartOut(TileInfo startOut) {
1474            throw new UnsupportedOperationException();
1475        }
1476
1477        @Override
1478        public void setEndOut(TileInfo endOut) {
1479            throw new UnsupportedOperationException();
1480        }
1481
1482        @Override
1483        public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
1484            throw new UnsupportedOperationException();
1485        }
1486
1487        @Override
1488        public void setCrateInside(boolean crateInside) {
1489            throw new UnsupportedOperationException();
1490        }
1491
1492        @Override
1493        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;
```

```
62              case LEFT -> UP;
63              case UP -> RIGHT;
64              case RIGHT -> DOWN;
65          };
66      }
67
68      /**
69       * @return The opposite direction (e.g for {@link Direction#LEFT} it returns {@link
    ↪  Direction#LEFT} etc.)
70       */
71      public Direction negate() {
72          return switch (this) {
73              case DOWN -> UP;
74              case UP -> DOWN;
75              case LEFT -> RIGHT;
76              case RIGHT -> LEFT;
77          };
78      }
79
80      /**
81       * Creates a direction from two coordinates.
82       * @param dirX If negative, returns {@link Direction#LEFT}, otherwise returns {@link
    ↪  Direction#RIGHT}
83       * @param dirY If negative, return {@link Direction#UP}, otherwise returns {@link Direction#DOWN}
84       * @return the direction
85       */
86      public static Direction of(int dirX, int dirY) {
87          if (dirX == 0 && dirY == 0) {
88              throw new IllegalArgumentException("(0,0) is not a direction");
89          } else if (dirX == 0) {
90              if (dirY < 0) {
91                  return UP;
92              } else {
93                  return DOWN;
94              }
95          } else if (dirX < 0) {
96              return LEFT;
97          } else {
98              return RIGHT;
99          }
100     }
101 }
```

GenericBoard

```
1  package fr.valax.sokoshell.solver.board;
2
3  import fr.valax.sokoshell.solver.Corral;
4  import fr.valax.sokoshell.solver.CorralDetector;
5  import fr.valax.sokoshell.solver.State;
6  import fr.valax.sokoshell.solver.board.tiles.Tile;
7  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
8
9  import java.util.function.Consumer;
10
11 /**
12  * A {@code package-private} class meant to be use as a base class for {@link Board} implementations.
13  * It defines all read-only methods, as well as a way to store the tiles. It is essentially a 2D-array
    ↪  of
14  * {@link TileInfo}, the indices being the y and x coordinates (i.e. {@code content[y][x]} is the tile
    ↪  at (x;y)).
15  *
```

```java
 * @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");
    }

```

```
79      @Override
80      public void addStateCrates(State state) {
81          throw new UnsupportedOperationException("Board is immutable");
82      }
83
84      @Override
85      public void removeStateCrates(State state) {
86          throw new UnsupportedOperationException("Board is immutable");
87      }
88
89      @Override
90      public void safeAddStateCrates(State state) {
91          throw new UnsupportedOperationException("Board is immutable");
92      }
93
94      @Override
95      public void safeRemoveStateCrates(State state) {
96          throw new UnsupportedOperationException("Board is immutable");
97      }
98
99      // Solver-used methods: throw UnsupportedOperationException as this object is (for now) not to be
        ↪  used by solvers //
100
101     @Override
102     public void initForSolver() {
103         throw new UnsupportedOperationException("Board is not intended for solvers");
104     }
105
106     @Override
107     public void computeFloors() {
108         throw new UnsupportedOperationException("Board is not intended for solvers");
109     }
110
111     @Override
112     public void forEachNotWall(Consumer<TileInfo> consumer) {
113         throw new UnsupportedOperationException("Board is not intended for solvers");
114     }
115
116     @Override
117     public void computeTunnelStatus(State state) {
118         throw new UnsupportedOperationException("Board is not intended for solvers");
119     }
120
121     @Override
122     public void computePackingOrderProgress(State state) {
123         throw new UnsupportedOperationException("Board is not intended for solvers");
124     }
125
126     @Override
127     public void computeDeadTiles() {
128         throw new UnsupportedOperationException("Board is not intended for solvers");
129     }
130
131     @Override
132     public void findTunnels() {
133         throw new UnsupportedOperationException("Board is not intended for solvers");
134     }
135
136     @Override
137     public void findRooms() {
138         throw new UnsupportedOperationException("Board is not intended for solvers");
139     }
140
```

```
141     @Override
142     public void tryComputePackingOrder() {
143         throw new UnsupportedOperationException("Board is not intended for solvers");
144     }
145 
146     @Override
147     public void findReachableCases(int playerPos) {
148         throw new UnsupportedOperationException("Board is not intended for solvers");
149     }
150 
151     @Override
152     public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
153         throw new UnsupportedOperationException("Board is not intended for solvers");
154     }
155 
156     @Override
157     public Corral getCorral(TileInfo tile) {
158         return null;
159     }
160 
161     @Override
162     public CorralDetector getCorralDetector() {
163         return null;
164     }
165 }
```

Board

```
1  package fr.valax.sokoshell.solver.board;
2  
3  import fr.valax.sokoshell.solver.Corral;
4  import fr.valax.sokoshell.solver.CorralDetector;
5  import fr.valax.sokoshell.solver.State;
6  import fr.valax.sokoshell.solver.board.mark.Mark;
7  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
8  import fr.valax.sokoshell.solver.board.tiles.Tile;
9  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
10 
11 import java.util.List;
12 import java.util.function.Consumer;
13 
14 /**
15  * Represents the Sokoban board.<br />
16  * This interface defines getters setters for the properties of a Sokoban board, e.g. the width, the
   ↪  height etc.
17  * Implementations of this interface are meant to be used with a {@link TileInfo} implementation.
18  * This class also defines static and dynamic analysis of the Sokoban board, for instance for solving
   ↪  purposes.
19  * Such properties are the following:
20  * <ul>
21  *     <li>Static</li>
22  *     <ul>
23  *         <li>Dead positions: cases that make the level unsolvable when a crate is pushed on
   ↪  them</li>
24  *     </ul>
25  *     <li>Dynamic</li>
26  *     <ul>
27  *         <li>Reachable cases: cases that the player can reach according to his position</li>
28  *     </ul>
29  * </ul>
30  *
31  * @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)
```

```
93       * @see #getY(int)
94       * @see #safeGetAt(int)
95       */
96      TileInfo getAt(int index);
97
98      /**
99       * Returns the {@link TileInfo} at the specific index
100      *
101      * @param index the index of the {@link TileInfo}
102      * @return the TileInfo at the specific index or {@code null}
103      * if the index represent a position outside the board
104      * @see #getX(int)
105      * @see #getY(int)
106      */
107     default TileInfo safeGetAt(int index) {
108         int x = getX(index);
109         int y = getY(index);
110
111         if (caseExists(x, y)) {
112             return getAt(x, y);
113         } else {
114             return null;
115         }
116     }
117
118     /**
119      * Returns the {@link TileInfo} at the specific position
120      *
121      * @param x x the of the tile
122      * @param y y the of the tile
123      * @return the TileInfo at the specific coordinate
124      * @throws IndexOutOfBoundsException if the position is outside the board
125      * @see #safeGetAt(int, int)
126      */
127     TileInfo getAt(int x, int y);
128
129     /**
130      * Returns the {@link TileInfo} at the specific position
131      *
132      * @param x x the of the tile
133      * @param y y the of the tile
134      * @return the TileInfo at the specific index or {@code null}
135      * if the index represent a position outside the board
136      * @see #getX(int)
137      * @see #getY(int)
138      */
139     default TileInfo safeGetAt(int x, int y) {
140         if (caseExists(x, y)) {
141             return getAt(x, y);
142         } else {
143             return null;
144         }
145     }
146
147     /**
148      * Tells whether the case at (x,y) exists or not (i.e. if the case is in the board)
149      *
150      * @param x x-coordinate
151      * @param y y-coordinate
152      * @return {@code true} if the case exists, {@code false} otherwise
153      */
154     default boolean caseExists(int x, int y) {
155         return (0 <= x && x < getWidth()) && (0 <= y && y < getHeight());
```

```
156     }
157
158     /**
159      * Same than caseExists(x, y) but with an index
160      *
161      * @param index index of the case
162      * @return {@code true} if the case exists, {@code false} otherwise
163      * @see #caseExists(int, int)
164      */
165     default boolean caseExists(int index) {
166         return caseExists(getX(index), getY(index));
167     }
168
169     /**
170      * Tells whether the tile at the given coordinates is empty or not.
171      *
172      * @param x x coordinate of the case
173      * @param y y coordinate of the case
174      * @return {@code true} if empty, {@code false} otherwise
175      */
176     default boolean isTileEmpty(int x, int y) {
177         TileInfo t = getAt(x, y);
178         return !t.isSolid();
179     }
180
181     /**
182      * Checks if the board is solved (i.e. all the crates are on a target).<br />
183      * <strong>The crates MUSTileInfo have been put on the board for this function to work as
    ↪ expected.</strong>
184      *
185      * @return {@code true} if the board is completed, false otherwise
186      */
187     default boolean isCompletedWith(State s) {
188         for (int i : s.cratesIndices()) {
189             if (!getAt(i).isCrateOnTarget()) {
190                 return false;
191             }
192         }
193         return true;
194     }
195
196     /**
197      * Checks if the board is completed (i.e. all the crates are on a target)
198      *
199      * @return true if completed, false otherwise
200      */
201     default boolean isCompleted() {
202         for (int y = 0; y < getHeight(); y++) {
203             for (int x = 0; x < getWidth(); x++) {
204                 if (getAt(x, y).isCrate()) {
205                     return false;
206                 }
207             }
208         }
209         return true;
210     }
211
212     /**
213      * Returns all tunnels that are in this board
214      *
215      * @return all tunnels that are in this board
216      */
217     List<Tunnel> getTunnels();
```

```
218
219     /**
220      * Returns all rooms that are in this board
221      *
222      * @return all rooms that are in this board
223      */
224     List<Room> getRooms();
225
226     boolean isGoalRoomLevel();
227
228     /**
228      * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking twice  a tile
230      *
231      * @return a mark system
232      * @see MarkSystem
233      */
234     MarkSystem getMarkSystem();
235
236     /**
237      * Returns the {@linkplain MarkSystem mark system} used by the {@link #findReachableCases(int)}
    ↪  algorithm
238      *
239      * @return the reachable mark system
240      * @see MarkSystem
241      */
242     MarkSystem getReachableMarkSystem();
243
244
245     // SETTERS //
246
247
248     /**
249      * Apply the consumer on every tile info
250      *
251      * @param consumer the consumer to apply
252      */
253     void forEach(Consumer<TileInfo> consumer);
254
255     /**
256      * Set at tile at the specified index. The index will be converted to
257      * cartesian coordinate with {@link #getX(int)} and {@link  #getY(int)}
258      *
259      * @param index index in the board
260      * @param tile  the new tile
261      * @throws IndexOutOfBoundsException if the index lead to a position outside the board
262      */
263     void setAt(int index, Tile tile);
264
265     /**
266      * Set at tile at (x, y)
267      *
268      * @param x x position in the board
269      * @param y y position in the board
270      * @throws IndexOutOfBoundsException if the position is outside the board
271      */
272     void setAt(int x, int y, Tile tile);
273
274     /**
275      * Puts the crates of the given state in the content array.
276      *
277      * @param state The state with the crates
278      */
279     void addStateCrates(State state);
```

```
280
281     /**
282      * Removes the crates of the given state from the content array.
283      *
284      * @param state The state with the crates
285      */
286     void removeStateCrates(State state);
287
288     /**
289      * Puts the crates of the given state in the content array.
290      * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
291      *
292      * @param state The state with the crates
293      */
294     void safeAddStateCrates(State state);
295
296     /**
297      * Removes the crates of the given state from the content array.
298      * If a crate is outside the board, it doesn't throw an {@link IndexOutOfBoundsException}
299      *
300      * @param state The state with the crates
301      */
302     void safeRemoveStateCrates(State state);
303
304     // ========================================
305     // *         Methods used by solvers         *
306     // * You need to call #initForSolver() first *
307     // ========================================
308
309     /**
310      * Initialize the board for solving:
311      * <ul>
312      *     <li>compute floor tiles: an array containing all non-wall tile</li>
313      *     <li>compute {@linkplain #computeDeadTiles() dead tiles}</li>
314      *     <li>find {@linkplain #findTunnels() tunnels}</li>
315      * </ul>
316      * <strong>The board must have no crate inside</strong>
317      *
318      * @see Tunnel
319      */
320     void initForSolver();
321
322     /**
323      * Creates or recreates the floor array. It is an array containing all tile info
324      * that are not a wall
325      */
326     void computeFloors();
327
328     /**
329      * Apply the consumer on every tile info except walls
330      *
331      * @param consumer the consumer to apply
332      */
333     void forEachNotWall(Consumer<TileInfo> consumer);
334
335     /**
336      * Compute which tunnel contains a crate
337      * @param state current state
338      */
339     void computeTunnelStatus(State state);
340
341     /**
342      * Compute packing order progress for each room if the level
```

```
343      * is a goal room level
344      * @param state current state
345      */
346     void computePackingOrderProgress(State state);
347 
348     // ************
349     // * ANALYSIS *
350     // ************
351 
352     // * STATIC *
353 
354     /**
355      * Detects the dead positions of a level. Dead positions are cases that make the level unsolvable
356      * when a crate is put on them.
357      * After this function has been called, to check if a given crate at (x,y) is a dead position,
358      * you can use {@link TileInfo#isDeadTile()} to check in constant time.
359      * The board <strong>MUST</strong> have <strong>NO CRATES</strong> for this function to work.
360      */
361     void computeDeadTiles();
362 
363     /**
364      * Find tunnels. A tunnel is something like this:
365      * <pre>
366      *     $$$$$$
367      *          $$$$$
368      *     $$$$
369      *        $$$$$$$
370      * </pre>
371      * <p>
372      * A tunnel doesn't contain a target
373      */
374     void findTunnels();
375 
376     /**
377      * Finds room based on tunnel. Basically all tile that aren't in a tunnel are in room.
378      * This means that you need to call {@link #findTunnels()} before!
379      * A room that contains a target is a packing room.
380      */
381     void findRooms();
382 
383     /**
384      * Compute packing order. No crate should be on the board
385      */
386     void tryComputePackingOrder();
387 
388     // * DYNAMIC *
389 
390     /**
391      * Find reachable tiles
392      *
393      * @param playerPos The indic of the case on which the player currently is.
394      */
395     void findReachableCases(int playerPos);
396 
397     /**
398      * This method compute the top left reachable position of the player of pushing a crate
399      * at crate to crateDest. It is used to calculate the position
400      * of the player in a {@link State}.
401      * This is also an example of use of {@link MarkSystem}
402      *
403      * @return the top left reachable position after pushing the crate
404      * @see MarkSystem
405      * @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;

/**
 * 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();
        }
```

```
45      }
46
47
48
49      protected final int playerPos;
50      protected final int[] cratesIndices;
51      protected final int hash;
52      protected final State parent;
53
54      public State(int playerPos, int[] cratesIndices, State parent) {
55          this(playerPos, cratesIndices, hashCode(playerPos, cratesIndices), parent);
56      }
57
58      public State(int playerPos, int[] cratesIndices, int hash, State parent) {
59          this.playerPos = playerPos;
60          this.cratesIndices = cratesIndices;
61          this.hash = hash;
62          this.parent = parent;
63      }
64
65      /**
66       * Creates a child of the state.
67       * It uses property of XOR to compute efficiently the hash of the child state
68       * @param newPlayerPos the new player position
69       * @param crateToMove the index of the crate to move
70       * @param crateDestination the new position of the crate to move
71       * @return the child state
72       */
73      public State child(int newPlayerPos, int crateToMove, int crateDestination) {
74          int[] newCrates = this.cratesIndices().clone();
75          int hash = this.hash ^ zobristValues[this.playerPos][0] ^ zobristValues[newPlayerPos][0] //
            ↪  'moves' the player in the hash
76                  ^ zobristValues[newCrates[crateToMove]][1] ^ zobristValues[crateDestination][1]; //
                    ↪  'moves' the crate in the hash
77          newCrates[crateToMove] = crateDestination;
78
79          return new State(newPlayerPos, newCrates, hash, this);
80      }
81
82      public long approxSizeOfAccurate() {
83          return SizeOf.getStateLayout().instanceSize() +
84                  SizeOf.getIntArrayLayout().instanceSize() +
85                  (long) Integer.BYTES * cratesIndices.length;
86      }
87
88      public long approxSizeOf() {
89          return 32 +
90                  16 +
91                  (long) Integer.BYTES * cratesIndices.length;
92      }
93
94      /**
95       * The index of the case of the board on which the player is.
96       */
97      public int playerPos() {
98          return playerPos;
99      }
100
101     /**
102      * The index of the cases of the board on which the crates are.
103      */
104     public int[] cratesIndices() {
105         return cratesIndices;
```

```
106     }
107
108     public int hash() {
109         return hash;
110     }
111
112     /**
113      * The state in which the board was before coming to this state.
114      */
115     public State parent() {
116         return parent;
117     }
118
119
120     @Override
121     public boolean equals(Object o) {
122         if (this == o) return true;
123         if (o == null || getClass() != o.getClass()) return false;
124
125         State state = (State) o;
126
127         if (playerPos != state.playerPos) return false;
128         return equals(cratesIndices, state.cratesIndices);
129     }
130
131     /**
132      * Returns true if all elements of array1 are included in array2 and vice-versa.
133      * However, because there is no duplicate and the two array have the same length,
134      * it is only necessary to check if array1 is included in array2.
135      *
136      * @param array1 the first array
137      * @param array2 the second array
138      * @return true if all elements are included in the second one
139      */
140     private boolean equals(int[] array1, int[] array2) {
141         for (int a : array1) {
142             if (!contains(a, array2)) {
143                 return false;
144             }
145         }
146
147         return true;
148     }
149
150     private boolean contains(int a, int[] array) {
151         for (int b : array) {
152             if (a == b) {
153                 return true;
154             }
155         }
156
157         return false;
158     }
159
160     @Override
161     public int hashCode() {
162         return hash;
163     }
164
165     public static int hashCode(int playerPos, int[] cratesIndices) {
166         int hash = zobristValues[playerPos][0];
167
168         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

```
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

```
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

```java
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 {

    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));
```

```
61              case RIGHT -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.y, player.getY() -
                ↪  t.x));
62              case DOWN -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.x, player.getY() - t.y));
63          };
64      }
65
66      private boolean isDeadlock(Function<DeadlockTable, TileInfo> getTile) {
67          if (deadlock == A_DEADLOCK) {
68              return true;
69          } else if (deadlock == NOT_A_DEADLOCK) {
70              return false;
71          }
72
73          TileInfo tile = getTile.apply(this);
74
75          if (tile == null) {
76              return false;
77          }
78
79          return switch (tile.getTile()) {
80              case FLOOR -> floorChild.isDeadlock(getTile);
81              case WALL -> wallChild.isDeadlock(getTile);
82              case CRATE -> crateChild.isDeadlock(getTile);
83              default -> false;
84          };
85      }
86
87      public static void write(DeadlockTable root, Path out) throws IOException {
88          try (OutputStream os = new BufferedOutputStream(Files.newOutputStream(out))) {
89              Stack<DeadlockTable> stack = new Stack<>();
90              stack.push(root);
91
92              while (!stack.isEmpty()) {
93                  DeadlockTable table = stack.pop();
94
95                  os.write(table.deadlock);
96                  if (table.deadlock == MAYBE_A_DEADLOCK) {
97                      writeInt(os, table.x);
98                      writeInt(os, table.y);
99                      stack.push(table.crateChild);
100                     stack.push(table.wallChild);
101                     stack.push(table.floorChild);
102                 }
103             }
104         }
105     }
106
107     public static DeadlockTable read(Path in) throws IOException {
108         try (InputStream is = new BufferedInputStream(Files.newInputStream(in))) {
109             return read(is);
110         }
111     }
112
113     private static DeadlockTable read(InputStream is) throws IOException {
114         int i = is.read();
115
116         if (i < 0 || i > 2) {
117             throw new IOException("Malformed table");
118         }
119
120         if (i == A_DEADLOCK) {
121             return DEADLOCK;
122         } else if (i == NOT_A_DEADLOCK) {
```

```
123             return NOT_DEADLOCK;
124         } else {
125             int x = readInt(is);
126             int y = readInt(is);
127 
128             DeadlockTable floor = read(is);
129             DeadlockTable wall = read(is);
130             DeadlockTable crate = read(is);
131 
132             return new DeadlockTable(MAYBE_A_DEADLOCK, x, y, floor, wall, crate);
133         }
134     }
135 
136     private static void writeInt(OutputStream os, int val) throws IOException {
137         os.write(val & 0xFF);
138         os.write((val >> 8) & 0xFF);
139         os.write((val >> 16) & 0xFF);
140         os.write((val >> 24) & 0xFF);
141     }
142 
143     private static int readInt(InputStream is) throws IOException {
144         int a = is.read() & 0xFF;
145         int b = is.read() & 0xFF;
146         int c = is.read() & 0xFF;
147         int d = is.read() & 0xFF;
148 
149         return (d << 24) | (c << 16) | (b << 8) | a;
150     }
151 
152     public static int countNotDetectedDeadlock(DeadlockTable table, int size) {
153         Board board = createBoard(size);
154 
155         // no dead tiles by default
156         board.setAt(1, 1, Tile.TARGET);
157         board.setAt(board.getWidth() - 2, board.getHeight() - 2, Tile.TARGET);
158 
159         board.computeFloors();
160         board.computeDeadTiles();
161         board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
162 
163         return countNotDetectedDeadlock(table, board, board.getWidth() / 2, board.getHeight() - 3);
164     }
165 
166     private static int countNotDetectedDeadlock(DeadlockTable table, Board board, int playerX, int
    ↪  playerY) {
167         if (table.deadlock == A_DEADLOCK) {
168             State state = createState(board, playerX, playerY);
169 
170             // but dead tiles aren't computed...
171             if (FreezeDeadlockDetector.checkFreezeDeadlock(board, state)) {
172                 return 0;
173             }
174 
175             CorralDetector detector = board.getCorralDetector();
176             detector.findCorral(board, playerX, playerY);
177             detector.findPICorral(board, state.cratesIndices());
178 
179             boolean deadlock = false;
180             for (Corral c : detector.getCorrals()) {
181                 if (c.isDeadlock(state, true)) {
182                     deadlock = true;
183                     break;
184                 }
```

```
185             }
186
187             if (deadlock) {
188                 return 0;
189             } else {
190                 BasicStyle.XSB_STYLE.print(board, playerX, playerY);
191
192                 return 1; // not detected !
193             }
194         } else if (table.deadlock == MAYBE_A_DEADLOCK) {
195             int n = countNotDetectedDeadlock(table.floorChild, board, playerX, playerY);
196
197             board.setAt(playerX + table.x, playerY + table.y, Tile.WALL);
198             n += countNotDetectedDeadlock(table.wallChild, board, playerX, playerY);
199
200             board.setAt(playerX + table.x, playerY + table.y, Tile.CRATE);
201             n += countNotDetectedDeadlock(table.crateChild, board, playerX, playerY);
202
203             board.setAt(playerX + table.x, playerY + table.y, Tile.FLOOR);
204
205             return n;
206         } else {
207             return 0;
208         }
209     }
210
211
212     public static DeadlockTable generate(int size) {
213         // if size = 3, returned board looks like:
214         // #######
215         // #     #
216         // #     #
217         // #     #
218         // #  @  #
219         // #######
220         // size of generated pattern: size * size
221         Board board = createBoard(size);
222
223         board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
224
225         return generate(board, createOrder(size), 0, board.getWidth() / 2, board.getHeight() - 3);
226     }
227
228     public static DeadlockTable generate2(int size, int nThread) {
229         Board board = createBoard(size);
230
231         board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
232
233         ForkJoinPool pool = new ForkJoinPool(nThread <= 0 ? Runtime.getRuntime().availableProcessors()
        ↪  :  nThread);
234         GenerateDeadlockTableTask task = new GenerateDeadlockTableTask(board, createOrder(size), 0,
        ↪  board.getWidth() / 2, board.getHeight() - 3, false);
235
236         DeadlockTable table = pool.invoke(task);
237         pool.shutdown();
238
239         return table;
240     }
241
242
243     private static DeadlockTable generate(Board board, int[][] order, int index, int playerX, int
    ↪  playerY) {
244         // BasicStyle.XSB_STYLE.print(board, playerX, playerY);
```

```
245
246         if (isDeadlock_(board, playerX, playerY)) {
247             return DEADLOCK;
248         } else if (index < order.length) {
249             int relativeX = order[index][0];
250             int relativeY = order[index][1];
251
252             board.setAt(playerX + relativeX, playerY + relativeY, Tile.WALL);
253             DeadlockTable wallChild = generate(board, order, index + 1, playerX, playerY);
254
255             board.setAt(playerX + relativeX, playerY + relativeY, Tile.CRATE);
256             DeadlockTable crateChild = generate(board, order, index + 1, playerX, playerY);
257
258             board.setAt(playerX + relativeX, playerY + relativeY, Tile.FLOOR);
259             if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
260                 return NOT_DEADLOCK;
261             }
262
263             DeadlockTable floorChild = generate(board, order, index + 1, playerX, playerY);
264
265             return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild, wallChild,
                ↪  crateChild);
266         } else {
267             return NOT_DEADLOCK;
268         }
269     }
270
271     private static Board createBoard(int size) {
272         Board board = new MutableBoard(size + 4, size + 4);
273         State.initZobristValues(board.getWidth() * board.getHeight());
274
275         for (int x = 0; x < board.getWidth(); x++) {
276             board.setAt(x, 0, Tile.WALL);
277             board.setAt(x, board.getHeight() - 1, Tile.WALL);
278         }
279
280         for (int y = 0; y < board.getHeight(); y++) {
281             board.setAt(0, y, Tile.WALL);
282             board.setAt(board.getWidth() - 1, y, Tile.WALL);
283         }
284
285         return board;
286     }
287
288     protected static int[][] createOrder(int size) {
289         int[][] order = new int[size * size - 2][2];
290
291         boolean odd = size % 2 == 1;
292         int i = 0;
293         int half = size / 2;
294         for (int y = 0; y > -size; y--) {
295             for (int x = -half; x < half || (x == half && odd); x++) {
296                 if (x == 0 && (y == 0 || y == -1)) {
297                     continue;
298                 }
299
300                 order[i] = new int[] {x, y};
301                 i++;
302             }
303         }
304
305
306         return order;
```

```
307     }
308
309     private static class GenerateDeadlockTableTask extends RecursiveTask<DeadlockTable> {
310
311         private static final AtomicInteger COUNTER = new AtomicInteger();
312         private static final int total = 4_782_969;
313
314         private final Board board;
315         private final int[][] order;
316         private final int index;
317         private final int playerX;
318         private final int playerY;
319         private final boolean check;
320
321         public GenerateDeadlockTableTask(Board board, int[][] order, int index, int playerX, int
            ↪  playerY, boolean check) {
322             this.board = board;
323             this.order = order;
324             this.index = index;
325             this.playerX = playerX;
326             this.playerY = playerY;
327             this.check = check;
328         }
329
330         @Override
331         protected DeadlockTable compute() {
332             int n = COUNTER.incrementAndGet();
333
334             if (n % 10_000 == 0) {
335                 System.out.printf("%.2f%% - %d%n", 100f * n / total, n);
336             }
337
338             if (check && isDeadlock_(board, playerX, playerY)) {
339                 return DEADLOCK;
340             } else if (index < order.length) {
341                 int relativeX = order[index][0];
342                 int relativeY = order[index][1];
343
344                 GenerateDeadlockTableTask wall = subTask(index, Tile.WALL, true);
345                 GenerateDeadlockTableTask crate = subTask(index, Tile.CRATE, true);
346
347                 wall.fork();
348                 crate.fork();
349
350                 DeadlockTable wallChild = wall.join();
351                 DeadlockTable crateChild = crate.join();
352
353                 if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
354                     return NOT_DEADLOCK;
355                 }
356
357                 GenerateDeadlockTableTask floor = subTask(index, Tile.FLOOR, false);
358                 DeadlockTable floorChild = floor.fork().join();
359
360                 // the three are never equals to deadlock because
361                 // it means the current board is a deadlock, and
362                 // it must be detected by isDeadlock_
363                 return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild,
                    ↪  wallChild, crateChild);
364
365             } else {
366                 return NOT_DEADLOCK;
367             }
```

```
368         }
369
370         private GenerateDeadlockTableTask subTask(int index, Tile replacement, boolean check) {
371             MutableBoard board = new MutableBoard(this.board);
372             int relativeX = order[index][0];
373             int relativeY = order[index][1];
374
375             board.setAt(playerX + relativeX, playerY + relativeY, replacement);
376
377             return new GenerateDeadlockTableTask(board, order, index + 1, playerX, playerY, check);
378         }
379     }
380
381
382
383
384
385     private static boolean isDeadlock_(Board board, int playerX, int playerY) {
386         State first = createState(board, playerX, playerY);
387
388         ReachableTiles reachableTiles = new ReachableTiles(board);
389         HashSet<State> visited = new HashSet<>();
390         Queue<State> toVisit = new ArrayDeque<>();
391
392         visited.add(first);
393         toVisit.offer(first);
394
395         boolean deadlock = true;
396         while (!toVisit.isEmpty() && deadlock) {
397             State parent = toVisit.poll();
398
399             board.addStateCrates(parent);
400
401             if (FreezeDeadlockDetector.checkFreezeDeadlock(board, parent)) {
402                 board.removeStateCrates(parent);
403                 continue;
404             }
405
406             reachableTiles.findReachableCases(board.getAt(parent.playerPos()));
407             deadlock = addChildrenStates(reachableTiles, parent, board, visited, toVisit);
408             board.removeStateCrates(parent);
409         }
410
411         board.addStateCrates(first);
412
413         return deadlock;
414     }
415
416     private static boolean addChildrenStates(ReachableTiles reachableTiles, State parent,
417                                              Board board, Set<State> visited, Queue<State> toVisit) {
418         for (int i = 0; i < parent.cratesIndices().length; i++) {
419             TileInfo crate = board.getAt(parent.cratesIndices()[i]);
420
421             for (Direction dir : Direction.VALUES) {
422                 TileInfo player = crate.adjacent(dir.negate());
423
424                 if (!reachableTiles.isReachable(player)) {
425                     continue;
426                 }
427
428                 TileInfo dest = crate.adjacent(dir);
429                 if (dest.isSolid()) {
430                     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;
```

```java
import fr.poulpogaz.json.IJsonWriter;
import fr.poulpogaz.json.JsonException;
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
```

```
66      protected void addParameters(List<SolverParameter> parameters) {
67          super.addParameters(parameters);
68          parameters.add(new HeuristicParameter());
69      }
70
71      @Override
72      public int lowerBound() {
73          return lowerBound;
74      }
75
76      protected static class HeuristicParameter extends SolverParameter {
77
78          private String value;
79
80          public HeuristicParameter() {
81              super("heuristic", "The heuristic the solver should use");
82          }
83
84          @Override
85          public void set(String argument) throws AbstractCommand.InvalidArgument {
86              if (argument.equalsIgnoreCase("greedy") || argument.equalsIgnoreCase("simple")) {
87                  this.value = argument;
88              } else {
89                  throw new AbstractCommand.InvalidArgument("No such heuristic: " + argument);
90              }
91          }
92
93          @Override
94          public Object get() {
95              return value;
96          }
97
98          @Override
99          public Object getDefaultValue() {
100             return "greedy";
101         }
102
103         @Override
104         public void toJson(IJsonWriter jw) throws JsonException, IOException {
105             jw.value(value);
106         }
107
108         @Override
109         public void fromJson(IJsonReader jr) throws JsonException, IOException {
110             value = jr.nextString();
111         }
112
113         @Override
114         public void complete(LineReader reader, String argument, List<Candidate> candidates) {
115             candidates.add(new Candidate("simple"));
116             candidates.add(new Candidate("greedy"));
117         }
118     }
119 }
```

Corral

```
1  package fr.valax.sokoshell.solver;
2
3  import fr.valax.sokoshell.solver.board.Board;
4  import fr.valax.sokoshell.solver.board.Direction;
5  import fr.valax.sokoshell.solver.board.Tunnel;
```

```java
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;
        }

        boolean deadlock = true;
        CorralState firstState = removeOutsideCrate(originalState);

        visited.add(firstState);
        toVisit.add(firstState);
```

```
69
70          while (!toVisit.isEmpty() && deadlock) {
71              currentState = toVisit.remove();
72
73              board.addStateCrates(currentState);
74
75              if (FreezeDeadlockDetector.checkFreezeDeadlock(board, currentState)) {
76                  board.removeStateCrates(currentState);
77                  continue;
78              }
79
80              board.computeTunnelStatus(currentState);
81              reachable.findReachableCases(board.getAt(currentState.playerPos()));
82              deadlock = addChildrenStates();
83
84              board.removeStateCrates(currentState);
85
86              if (visited.size() >= 1000) {
87                  deadlock = false;
88              }
89          }
90
91          visited.clear();
92          toVisit.clear();
93
94          // re-add crates
95          board.addStateCrates(originalState);
96
97          return deadlock;
98      }
99
100     private void addFrozenCrates(State state) {
101         for (int i : state.cratesIndices) {
102             TileInfo crate = board.getAt(i);
103
104             if (crates.contains(crate)) {
105                 continue;
106             }
107
108             if (isFrozen(crate, Direction.LEFT) && isFrozen(crate, Direction.UP)) {
109                 crates.add(crate);
110             }
111         }
112     }
113
114     /**
115      * True if the crate is almost frozen ie right now it can be moved
116      * in the axis: it happens when an adjacent tile on the axis is solid.
117      * The adjacent tile must be in the corral is it is a crate
118      */
119     private boolean isFrozen(TileInfo tile, Direction axis) {
120         TileInfo left = tile.adjacent(axis);
121         TileInfo right = tile.adjacent(axis.negate());
122
123         return left.isWall() ||
124                 left.anyCrate() && crates.contains(left) ||
125                 right.isWall() ||
126                 right.anyCrate() && crates.contains(right);
127     }
128
129
130     /**
131      * @return false if not a deadlock
```

```
132      */
133     private boolean addChildrenStates() {
134         int[] cratesIndices = currentState.cratesIndices();
135 
136         boolean deadlock = true;
137         for (int i = 0; i < cratesIndices.length && deadlock; i++) {
138             TileInfo crate = board.getAt(cratesIndices[i]);
139 
140             if (crate.isInATunnel()) {
141                 deadlock = addChildrenStatesInTunnel(i, crate);
142             } else {
143                 deadlock = addChildrenStatesDefault(i, crate);
144             }
145         }
146 
147         return deadlock;
148     }
149 
150     //
151     // THE TWO FOLLOWING METHODS ARE COPIED FROM ABSTRACT SOLVER.
152     // I hope that one day, I will change that
153     //
154 
155     protected boolean addChildrenStatesInTunnel(int crateIndex, TileInfo crate) {
156         // the crate is in a tunnel. two possibilities: move to tunnel.startOut or tunnel.endOut
157         // this part of the code assume that there is no other crate in the tunnel.
158         // normally, this is impossible...
159 
160         for (Direction pushDir : Direction.VALUES) {
161             TileInfo player = crate.adjacent(pushDir.negate());
162 
163             if (reachable.isReachable(player)) {
164                 TileInfo dest = crate.getTunnelExit().getExit(pushDir);
165 
166                 if (dest != null && !dest.isSolid()) {
167                     if (!addState(crateIndex, crate, dest, pushDir)) {
168                         return false; // not a deadlock
169                     }
170                 }
171             }
172         }
173 
174         return true;
175     }
176 
177     protected boolean addChildrenStatesDefault(int crateIndex, TileInfo crate) {
178         for (Direction d : Direction.VALUES) {
179             TileInfo crateDest = crate.adjacent(d);
180             if (crateDest.isSolid()) {
181                 continue; // The destination case is not empty
182             }
183 
184             if (crateDest.isDeadTile()) {
185                 continue; // Useless to push a crate on a dead position
186             }
187 
188             TileInfo player = crate.adjacent(d.negate());
189             if (!reachable.isReachable(player)) {
190                 // The player cannot reach the case to push the crate
191                 // also checks if tile is solid: a solid tile is never reachable
192                 continue;
193             }
194 
```

```
195
196             // check for tunnel
197             Tunnel tunnel = crateDest.getTunnel();
198
199             // the crate will be pushed inside the tunnel
200             if (tunnel != null) {
201                 if (tunnel.crateInside()) { // pushing inside will lead to a corral deadlock
202                     continue;
203                 }
204
205                 // ie the crate can't be pushed to the other extremities of the tunnel
206                 // however, sometimes (boxxle 24) it is useful to push the crate inside
207                 // the tunnel. That's why the second addState is done (after this if)
208                 // and only if this tunnel isn't oneway
209                 if (!tunnel.isPlayerOnlyTunnel()) {
210                     TileInfo newDest = null;
211                     Direction pushDir = null;
212
213                     if (crate == tunnel.getStartOut()) {
214                         if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate()) {
215                             newDest = tunnel.getEndOut();
216                             pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
217                         }
218                     } else {
219                         if (tunnel.getStartOut() != null && !tunnel.getStartOut().anyCrate()) {
220                             newDest = tunnel.getStartOut();
221                             pushDir = tunnel.getStart().direction(tunnel.getStartOut());
222                         }
223                     }
224
225                     if (newDest != null && !newDest.isDeadTile()) {
226                         if (!addState(crateIndex, crate, newDest, pushDir)) {
227                             return false;
228                         }
229                     }
230                 }
231
232                 if (tunnel.isOneway()) {
233                     continue;
234                 }
235             }
236
237             if (!addState(crateIndex, crate, crateDest, d)) {
238                 return false;
239             }
240         }
241
242         return true;
243     }
244
245     /**
246      * @return false if not a deadlock
247      */
248     private boolean addState(int crateIndex, TileInfo crate, TileInfo dest, Direction pushDir) {
249         // a crate can be moved outside the corral
250         if (!isInCorral(dest)) {
251             return false;
252         }
253
254         if (deadlockTable.isDeadlock(crate.adjacent(pushDir.negate()), pushDir)) {
255             return true; // current state is a deadlock, we need to continue the research
256         }
257
```

```
258         // all crates of the corral can be moved to a target
259 
260         int n = 0;
261         for (int i : currentState.cratesIndices()) {
262             if (i != crate.getIndex() && board.getAt(i).isCrateOnTarget()) {
263                 n++;
264             }
265         }
266 
267         if (dest.isTarget() && n + 1 == currentState.cratesIndices.length) { // TODO: crate may be on
        ↪  target
268             return false;
269         }
270 
271         // create sub state
272         int newPlayerPos = board.topLeftReachablePosition(crate, dest);
273         CorralState sub = currentState.child(newPlayerPos, crateIndex, dest.getIndex());
274 
275         if (crate.isCrate() && dest.isTarget()) {
276             sub.increaseNumberOnTarget();
277         } else if (crate.isCrateOnTarget() && dest.isFloor()) {
278             sub.decreaseNumberOnTarget();
279         }
280 
281         if (visited.add(sub)) {
282             toVisit.offer(sub);
283         }
284 
285         return true;
286     }
287 
288     /**
289      * Remove crates that are not part of the corral
290      * and create a new state without these crates
291      * @param state current state
292      * @return a state without crate outside the corral
293      */
294     private CorralState removeOutsideCrate(State state) {
295         int numOnTarget = 0;
296 
297         int[] newCrates = new int[crates.size()];
298         int[] oldCrates = state.cratesIndices();
299         int j = 0;
300         for (int i = 0; i < oldCrates.length; i++) {
301             TileInfo crate = board.getAt(oldCrates[i]);
302             if (isInCorral(oldCrates[i])) {
303                 if (crate.isCrateOnTarget()) {
304                     numOnTarget++;
305                 }
306 
307                 newCrates[j] = oldCrates[i];
308                 j++;
309             } else {
310                 crate.removeCrate();
311             }
312         }
313 
314         CorralState corralState = new CorralState(state.playerPos(), newCrates, null);
315         corralState.setNumOnTarget(numOnTarget);
316         return corralState;
317     }
318 
319     private boolean isInCorral(int crate) {
```

```
320         TileInfo tile = board.getAt(crate);
321
322         return crates.contains(tile);
323     }
324
325     private boolean isInCorral(TileInfo tile) {
326         Corral c = board.getCorral(tile);
327
328         if (c == null) {
329             return isInCorral(tile.getIndex());
330         } else {
331             return c == this;
332         }
333     }
334
335     public int getTopX() {
336         return topX;
337     }
338
339     public int getTopY() {
340         return topY;
341     }
342
343     public List<TileInfo> getBarrier() {
344         return barrier;
345     }
346
347     public List<TileInfo> getCrates() {
348         return crates;
349     }
350
351     public boolean containsPlayer() {
352         return containsPlayer;
353     }
354
355     public boolean isPICorral() {
356         return isPICorral == IS_A_PI_CORRAL;
357     }
358
359     public DeadlockTable getDeadlockTable() {
360         return deadlockTable;
361     }
362
363     public void setDeadlockTable(DeadlockTable deadlockTable) {
364         this.deadlockTable = deadlockTable;
365     }
366
367     @Override
368     public int hashCode() {
369         return id;
370     }
371
372     @Override
373     public boolean equals(Object o) {
374         if (this == o) return true;
375         if (!(o instanceof Corral corral)) return false;
376
377         return id == corral.id;
378     }
379
380     private static class CorralState extends State {
381
382         private int numOnTarget;
```

```
383
384         public CorralState(int playerPos, int[] cratesIndices, State parent) {
385             super(playerPos, cratesIndices, parent);
386         }
387
388         public CorralState(int playerPos, int[] cratesIndices, int hash, State parent) {
389             super(playerPos, cratesIndices, hash, parent);
390         }
391
392         private CorralState(State state) {
393             super(state.playerPos, state.cratesIndices, state.hash, state.parent);
394         }
395
396         @Override
397         public CorralState child(int newPlayerPos, int crateToMove, int crateDestination) {
398             return new CorralState(super.child(newPlayerPos, crateToMove, crateDestination));
399         }
400
401         public void increaseNumberOnTarget() {
402             numOnTarget++;
403         }
404
405         public void decreaseNumberOnTarget() {
406             numOnTarget--;
407         }
408
409         public int getNumOnTarget() {
410             return numOnTarget;
411         }
412
413         public void setNumOnTarget(int numOnTarget) {
414             this.numOnTarget = numOnTarget;
415         }
416     }
417 }
```

BruteforceSolver

```
1  package fr.valax.sokoshell.solver;
2
3  import fr.valax.sokoshell.solver.board.Direction;
4  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
5  import fr.valax.sokoshell.solver.collections.SolverCollection;
6
7  import java.util.ArrayDeque;
8
9  /**
10  * This class serves as a base class for DFS and BFS solvers, as these class are nearly the same --
    ↪  the only
11  * difference being in the order in which they treat the states (LIFO for DFS and FIFO for BFS).
12  */
13 public abstract class BruteforceSolver extends AbstractSolver<State> {
14
15     public BruteforceSolver(String name) {
16         super(name);
17     }
18
19     public static DFSSolver newDFSSolver() {
20         return new DFSSolver();
21     }
22
23     public static BFSSolver newBFSSolver() {
```

```
24          return new BFSSolver();
25      }
26
27      @Override
28      protected void addInitialState(Level level) {
29          toProcess.addState(level.getInitialState());
30      }
31
32      @Override
33      protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
34          if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
35              return;
36          }
37
38          final int i = board.topLeftReachablePosition(crate, crateDest);
39          // The new player position is the crate position
40          State s = toProcess.cachedState().child(i, crate.getCrateIndex(), crateDest.getIndex());
41
42          if (processed.add(s)) {
43              toProcess.addState(s);
44          }
45      }
46
47      @Override
48      public int lowerBound() {
49          return -1;
50      }
51
52      /**
53       * Base class for DFS and BFS solvers collection (both of them use {@link ArrayDeque}), the only
    ↪ difference being in
54       * which side of the queue is used (end => FIFO => DFS, start => LIFO => BFS)
55       */
56      private static abstract class BasicBruteforceSolverCollection implements SolverCollection<State> {
57
58          protected final ArrayDeque<State> collection = new ArrayDeque<>();
59
60          protected State cachedState;
61
62          @Override
63          public void clear() {
64              collection.clear();
65          }
66
67          @Override
68          public boolean isEmpty() {
69              return collection.isEmpty();
70          }
71
72          @Override
73          public int size() {
74              return collection.size();
75          }
76
77          @Override
78          public void addState(State state) {
79              collection.offer(state);
80          }
81
82          @Override
83          public State peekAndCacheState() {
84              cachedState = popState();
85              return cachedState;
```

```
86          }
87
88          @Override
89          public State cachedState() {
90              return cachedState;
91          }
92      }
93
94      private static class DFSSolver extends BruteforceSolver {
95
96          public DFSSolver() {
97              super(DFS);
98          }
99
100         @Override
101         protected void init(SolverParameters parameters) {
102             toProcess = new DFSSolverCollection();
103         }
104
105         private static class DFSSolverCollection extends BasicBruteforceSolverCollection {
106
107             @Override
108             public State popState() {
109                 return collection.removeLast();
110             }
111
112             @Override
113             public State peekState() {
114                 return collection.peekLast();
115             }
116         }
117     }
118
119     private static class BFSSolver extends BruteforceSolver {
120
121         public BFSSolver() {
122             super(BFS);
123         }
124
125         @Override
126         protected void init(SolverParameters parameters) {
127             toProcess = new BFSSolverCollection();
128         }
129
130         private static class BFSSolverCollection extends BasicBruteforceSolverCollection {
131
132             @Override
133             public State popState() {
134                 return collection.removeFirst();
135             }
136
137             @Override
138             public State peekState() {
139                 return collection.peekFirst();
140             }
141
142         }
143     }
144 }
```

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 {
```

```
20
21      protected static final String TIMEOUT = "timeout";
22      protected static final String MAX_RAM = "max-ram";
23      protected static final String ACCURATE = "accurate";
24
25      protected final String name;
26
27      protected final DeadlockTable table;
28
29      protected SolverCollection<S> toProcess;
30      protected final Set<State> processed = new HashSet<>();
31
32      protected MutableBoard board;
33
34      private boolean running = false;
35      private boolean stopped = false;
36
37      // statistics
38      private long timeStart = -1;
39      private long timeEnd = -1;
40      private int nStateProcessed = -1;
41      private int queueSize = -1;
42      private Tracker tracker;
43
44      public AbstractSolver(String name) {
45          this.name = name;
46
47          try {
48              table = DeadlockTable.read(Path.of("4x4.table"));
49          } catch (IOException e) {
50              throw new RuntimeException(e);
51          }
52      }
53
54      @Override
55      public SolverReport solve(SolverParameters params) {
56          Objects.requireNonNull(params);
57
58          // init statistics, timeout and stop
59          String endStatus = null;
60
61          running = true;
62          stopped = false;
63
64          long timeout = params.getArgument(TIMEOUT);
65          long maxRam = params.getArgument(MAX_RAM);
66          boolean accurate = params.getArgument(ACCURATE);
67
68          if (accurate) {
69              SizeOf.initialize();
70          }
71
72          timeStart = System.currentTimeMillis();
73          timeEnd = -1;
74          nStateProcessed = 0;
75          queueSize = 0;
76
77          if (tracker != null) {
78              tracker.reset();
79          }
80
81          // init the research
82
```

```
        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;
            }

            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();
```

```
146         queueSize = toProcess.size();
147 
148         // 'free' ram
149         processed.clear();
150         toProcess.clear();
151         board = null;
152 
153         running = false;
154 
155         System.out.println("END: " + finalState + " - " + endStatus);
156 
157         if (endStatus != null) {
158             return SolverReport.withoutSolution(params, getStatistics(), endStatus);
159         } else if (stopped) {
160             return SolverReport.withoutSolution(params, getStatistics(), SolverReport.STOPPED);
161         } else if (finalState != null) {
162             return SolverReport.withSolution(finalState, params, getStatistics());
163         } else {
164             return SolverReport.withoutSolution(params, getStatistics(), SolverReport.NO_SOLUTION);
165         }
166     }
167 
168     /**
169      * Initialize the solver. This method is called after the initialization of
170      * the board
171      */
172     protected abstract void init(SolverParameters parameters);
173 
174     protected abstract void addInitialState(Level level);
175 
176     protected boolean checkPICorralDeadlock(State state) {
177         CorralDetector detector = board.getCorralDetector();
178         detector.findPICorral(board, state.cratesIndices());
179 
180         for (Corral corral : detector.getCorrals()) {
181             if (corral.isDeadlock(state)) {
182                 return true;
183             }
184         }
185 
186         return false;
187     }
188 
189     protected void addChildrenStates(TileInfo player) {
190         Corral playerCorral = board.getCorralDetector().findCorral(player);
191 
192         List<TileInfo> crates = playerCorral.getCrates();
193         for (int i = 0; i < crates.size(); i++) {
194             TileInfo crateTile = crates.get(i);
195 
196             // check if the crate is already at his destination
197             if (board.isGoalRoomLevel() && crateTile.isInARoom()) {
198                 Room r = crateTile.getRoom();
199 
200                 if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
201                     continue;
202                 } else {
203                     tryGoalCut(crateTile);
204                 }
205             }
206 
207             Tunnel tunnel = crateTile.getTunnel();
208             if (tunnel != null) {
```

```
209                 addChildrenStatesInTunnel(crateTile);
210             } else {
211                 addChildrenStatesDefault(crateTile);
212             }
213         }
214     }
215 
216     protected void tryGoalCut(TileInfo crate) {
217         TileInfo player = board.getAt(currentState().playerPos());
218 
219         // only works because rooms have one entry
220         CrateAStar crateAStar = board.getCrateAStar();
221         List<Room> rooms = board.getRooms();
222         for (int i = 0; i < rooms.size(); i++) {
223             Room r = rooms.get(i);
224 
225             Tunnel tunnel = r.getTunnels().get(0);
226             TileInfo entrance;
227             if (tunnel.getStartOut().getRoom() == r) {
228                 entrance = tunnel.getStartOut();
229             } else {
230                 entrance = tunnel.getEndOut();
231             }
232 
233             if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
234                 if (crateAStar.hasPath(player, null, crate, entrance)) {
235                     addStateCheckForGoalMacro(crate, entrance, null);
236                 }
237             }
238         }
239     }
240 
241     protected void addChildrenStatesInTunnel(TileInfo crate) {
242         // the crate is in a tunnel. two possibilities: move to tunnel.startOut or tunnel.endOut
243         // this part of the code assume that there is no other crate in the tunnel.
244         // normally, this is impossible...
245 
246         for (Direction pushDir : Direction.VALUES) {
247             TileInfo player = crate.adjacent(pushDir.negate());
248 
249             if (player.isReachable()) {
250                 TileInfo dest = crate.getTunnelExit().getExit(pushDir);
251 
252                 if (dest != null && !dest.isSolid()) {
253                     addStateCheckForGoalMacro(crate, dest, pushDir);
254                 }
255             }
256         }
257     }
258 
259     protected void addChildrenStatesDefault(TileInfo crate) {
260         for (Direction d : Direction.VALUES) {
261 
262             TileInfo crateDest = crate.adjacent(d);
263             if (crateDest.isSolid()) {
264                 continue; // The destination case is not empty
265             }
266 
267             if (crateDest.isDeadTile()) {
268                 continue; // Useless to push a crate on a dead position
269             }
270 
271             TileInfo player = crate.adjacent(d.negate());
```

```
272             if (!player.isReachable()) {
273                 // The player cannot reach the case to push the crate
274                 // also checks if tile is solid: a solid tile is never reachable
275                 continue;
276             }
277 
278 
279             // check for tunnel
280             Tunnel tunnel = crateDest.getTunnel();
281 
282             // the crate will be pushed inside the tunnel
283             if (tunnel != null) {
284                 if (tunnel.crateInside()) { // pushing inside will lead to a corral deadlock
285                     continue;
286                 }
287 
288                 // ie the crate can't be pushed to the other extremities of the tunnel
289                 // however, sometimes (boxxle 24) it is useful to push the crate inside
290                 // the tunnel. That's why the second addState is done (after this if)
291                 // and only if this tunnel isn't oneway
292                 if (!tunnel.isPlayerOnlyTunnel()) {
293                     TileInfo newDest = null;
294                     Direction pushDir = null;
295 
296                     if (crate == tunnel.getStartOut()) {
297                         if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate()) {
298                             newDest = tunnel.getEndOut();
299                             pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
300                         }
301                     } else {
302                         if (tunnel.getStartOut() != null && !tunnel.getStartOut().anyCrate()) {
303                             newDest = tunnel.getStartOut();
304                             pushDir = tunnel.getStart().direction(tunnel.getStartOut());
305                         }
306                     }
307 
308                     if (newDest != null && !newDest.isDeadTile()) {
309                         addStateCheckForGoalMacro(crate, newDest, pushDir);
310                     }
311                 }
312 
313                 if (tunnel.isOneway()) {
314                     continue;
315                 }
316             }
317 
318             addStateCheckForGoalMacro(crate, crateDest, d);
319         }
320     }
321 
322     protected void addStateCheckForGoalMacro(TileInfo crate, TileInfo dest, Direction pushDir) {
323         Room room = dest.getRoom();
324         if (room != null && board.isGoalRoomLevel() && room.getPackingOrderIndex() >= 0) {
325             // goal macro!
326             TileInfo newDest = room.getPackingOrder().get(room.getPackingOrderIndex());
327 
328             addState(crate, newDest, null);
329         } else {
330             addState(crate, dest, pushDir);
331         }
332     }
333 
334     /**
```

```
335      * Check if the move leads to a deadlock.
336      * Only for simple deadlock that don't require
337      * lots of computation like PI Corral deadlock
338      *
339      * @param crate crate to move
340      * @param crateDest crate destination
341      * @param pushDir push dir of the player. If the move is a macro move,
342      *                it is the last push done by the player. It can be null
343      * @return true if deadlock
344      */
345     protected boolean checkDeadlockBeforeAdding(TileInfo crate, TileInfo crateDest, Direction pushDir)
         ↪ {
346         crate.removeCrate();
347         crateDest.addCrate();
348 
349         boolean deadlock = FreezeDeadlockDetector.checkFreezeDeadlock(crateDest);
350 
351         if (!deadlock && pushDir != null) {
352             deadlock = table.isDeadlock(crateDest.adjacent(pushDir.negate()), pushDir);
353         }
354 
355         crate.addCrate();
356         crateDest.removeCrate();
357 
358         return deadlock;
359     }
360 
361     /**
362      * Add a state to the processed set. If it wasn't already added, it is added to
363      * the toProcess queue. The move is unchecked
364      *
365      * @param crate crate to move
366      * @param crateDest crate destination
367      * @param pushDir push dir of the player. If the move is a macro move,
368      *                it is the last push done by the player. It can be null
369      */
370     protected abstract void addState(TileInfo crate, TileInfo crateDest, Direction pushDir);
371 
372     protected boolean hasTimedOut(long timeout) {
373         return timeout > 0 && timeout + timeStart < System.currentTimeMillis();
374     }
375 
376     protected boolean hasRamExceeded(long maxRam, boolean accurate) {
377         if (maxRam > 0) {
378             State curr = currentState();
379 
380             if (curr != null) {
381                 long stateSize;
382                 long ramUsed;
383                 if (accurate) {
384                     stateSize = curr.approxSizeOfAccurate();
385                     ramUsed = SizeOf.approxSizeOfAccurate(processed, stateSize);
386                 } else {
387                     stateSize = curr.approxSizeOf();
388                     ramUsed = SizeOf.approxSizeOf(processed, stateSize);
389                 }
390 
391                 return ramUsed + toProcess.size() * stateSize >= maxRam;
392             }
393         }
394 
395         return false;
396     }
```

```
397
398     @Override
399     public String getName() {
400         return name;
401     }
402
403     @Override
404     public boolean isRunning() {
405         return running;
406     }
407
408     @Override
409     public boolean stop() {
410         stopped = true;
411         return true;
412     }
413
414
415     @Override
416     public List<SolverParameter> getParameters() {
417         List<SolverParameter> params = new ArrayList<>();
418         addParameters(params);
419         return params;
420     }
421
422     /**
423      * Add your parameters to the list returned by {@link #getParameters()}
424      * @param parameters parameters that will be returned by {@link #getParameters()}
425      */
426     protected void addParameters(List<SolverParameter> parameters) {
427         parameters.add(new SolverParameter.Long(TIMEOUT, "Maximal runtime of the solver", -1));
428         parameters.add(new SolverParameter.RamParameter(MAX_RAM, -1));
429         parameters.add(new SolverParameter.Boolean(ACCURATE,
              ↪  "Use a more accurate method to calculate ram usage", false));
430     }
431
432     private ISolverStatistics getStatistics() {
433         ISolverStatistics stats;
434
435         if (tracker != null) {
436             stats = Objects.requireNonNull(tracker.getStatistics(this));
437         } else {
438             stats = new ISolverStatistics.Basic(timeStart, timeEnd);
439         }
440
441         return stats;
442     }
443
444     @Override
445     public State currentState() {
446         if (toProcess != null && running) {
447             return toProcess.cachedState();
448         } else {
449             return null;
450         }
451     }
452
453     @Override
454     public Board staticBoard() {
455         if (board != null && running) {
456             return board.staticBoard();
457         } else {
458             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;
        }
    }

    @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

```
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;
```

```java
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;


```

```
77
78
79      public static class Integer extends SolverParameter {
80
81          protected final int defaultValue;
82          protected java.lang.Integer value = null;
83
84          public Integer(String name, int defaultValue) {
85              this(name, null, defaultValue);
86          }
87
88          public Integer(String name, String description, int defaultValue) {
89              super(name, description);
90              this.defaultValue = defaultValue;
91          }
92
93          @Override
94          public void set(String argument) throws AbstractCommand.InvalidArgument {
95              try {
96                  value = java.lang.Integer.parseInt(argument);
97              } catch (NumberFormatException e) {
98                  throw new AbstractCommand.InvalidArgument(e);
99              }
100         }
101
102         @Override
103         public Object get() {
104             return value;
105         }
106
107         @Override
108         public Object getDefaultValue() {
109             return defaultValue;
110         }
111
112         @Override
113         public void toJson(IJsonWriter jw) throws JsonException, IOException {
114             if (value != null) {
115                 jw.value(value);
116             }
117         }
118
119         @Override
120         public void fromJson(IJsonReader jr) throws JsonException, IOException {
121             value = jr.nextInt();
122         }
123     }
124
125     public static class Long extends SolverParameter {
126
127         protected final long defaultValue;
128         protected java.lang.Long value = null;
129
130         public Long(String name, long defaultValue) {
131             this(name, null, defaultValue);
132         }
133
134         public Long(String name, String description, long defaultValue) {
135             super(name, description);
136             this.defaultValue = defaultValue;
137         }
138
139         @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
```

```
203         public Object getDefaultValue() {
204             return defaultValue;
205         }
206 
207         @Override
208         public void toJson(IJsonWriter jw) throws JsonException, IOException {
209             if (value != null) {
210                 jw.value(value);
211             }
212         }
213 
214         @Override
215         public void fromJson(IJsonReader jr) throws JsonException, IOException {
216             value = jr.nextBoolean();
217         }
218     }
219 
220 
221 
222     public static class RamParameter extends Long {
223 
224         private static final Pattern PATTERN = Pattern.compile("^(\\d+)\\s*([gmk])?b$",
        ↪  Pattern.CASE_INSENSITIVE);
225 
226         public RamParameter(String name, long defaultValue) {
227             super(name, "Maximal ram usage of the solver", defaultValue);
228         }
229 
230         public RamParameter(String name, String description, long defaultValue) {
231             super(name, description, defaultValue);
232         }
233 
234         @Override
235         public void set(String argument) throws AbstractCommand.InvalidArgument {
236             Matcher matcher = PATTERN.matcher(argument);
237 
238             if (matcher.matches() && matcher.groupCount() >= 1 && matcher.groupCount() <= 2) {
239                 long r = java.lang.Long.parseLong(matcher.group(1));
240 
241                 if (matcher.groupCount() == 2) {
242                     String unit = matcher.group(2).toLowerCase();
243 
244                     r = switch (unit) {
245                         case "g" -> r * 1024 * 1024 * 1024;
246                         case "m" -> r * 1024 * 1024;
247                         case "k" -> r * 1024;
248                         default -> throw new AbstractCommand.InvalidArgument("Invalid ram argument");
249                     };
250                 }
251 
252                 value = r;
253             } else {
254                 throw new AbstractCommand.InvalidArgument("Invalid ram argument");
255             }
256         }
257     }
258 }
```

Trackable

```
1 package fr.valax.sokoshell.solver;
2 
```

```java
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
     */
```

```
66      Tracker getTracker();
67  }
```

CorralDetector

```
1   package fr.valax.sokoshell.solver;
2
3   import fr.valax.sokoshell.solver.board.Board;
4   import fr.valax.sokoshell.solver.board.Direction;
5   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
7   import java.util.*;
8
9   /**
10   * A union find structure to find corral in a map.
11   * The objective of this object is to compute corral,
12   * barriers and topY, topX position of each corral.
13   */
14  @SuppressWarnings("ForLoopReplaceableByForEach")
15  public class CorralDetector {
16
17      private final Corral[] corrals;
18      private final int[] parent;
19      private final int[] rank;
20
21      private final Set<Corral> currentCorrals;
22
23      private int realNumberOfCorral;
24
25      public CorralDetector(Board board) {
26          int size = board.getWidth() * board.getHeight();
27          parent = new int[size];
28          rank = new int[size];
29          corrals = new Corral[size];
30
31          for (int i = 0; i < parent.length; i++) {
32              parent[i] = i;
33              corrals[i] = new Corral(i, board);
34          }
35
36          currentCorrals = new HashSet<>(size);
37      }
38
39      /**
40       * Find corral. Compute topX, topY. Find the corral that
41       * contains the player.
42       * Other values (isPICorral, crates, barriers) are not
43       * valid after a call to this method. Use {@link #findPICorral(Board, int[])}
44       * to revalidate them.
45       *
46       * @param board the board
47       * @param playerX player position x
48       * @param playerY player position y
49       */
50      public void findCorral(Board board, int playerX, int playerY) {
51          currentCorrals.clear();
52
53          int h = board.getHeight();
54          int w = board.getWidth();
55
56          for (int y = 1; y < h - 1; y++) {
57              TileInfo left = board.getAt(0, y);
```

```
58
59              for (int x = 1; x < w - 1; x++) {
60                  TileInfo t = board.getAt(x, y);
61
62                  if (!t.isSolid()) {
63                      TileInfo up = board.getAt(x, y - 1);
64
65                      if (!up.isSolid() && !left.isSolid()) {
66                          addToCorral(t, up);
67                          mergeTwoCorrals(up, left);
68                      } else if (!up.isSolid()) {
69                          addToCorral(t, up);
70                      } else if (!left.isSolid()) {
71                          addToCorral(t, left);
72                      } else {
73                          newCorral(t);
74                      }
75                  } else {
76                      int i = t.getIndex();
77                      parent[i] = -1;
78                      rank[i] = -1;
79                      corrals[i].isValid = false;
80                  }
81
82                  left = t;
83              }
84          }
85
86          int playerCorral = find(playerY * board.getWidth() + playerX);
87          corrals[playerCorral].containsPlayer = true;
88
89          realNumberOfCorral = currentCorrals.size();
90      }
91
92      /**
93       * Find PI corral
94       * @param board the board
95       * @param crates crates on the board
96       */
97      public void findPICorral(Board board, int[] crates) {
98          preComputePICorral(board, crates);
99
100         List<Corral> corrals = new ArrayList<>(currentCorrals);
101
102         for (int i = 0; i < corrals.size(); i++) {
103             Corral c = corrals.get(i);
104
105             if (!c.containsPlayer()) {
106                 if (isPICorral(c)) {
107                     c.isPICorral = Corral.IS_A_PI_CORRAL;
108                     corrals.remove(i);
109                     i--;
110                 }
111             } else {
112                 c.isPICorral = Corral.NOT_A_PI_CORRAL;
113                 corrals.remove(i);
114                 i--;
115             }
116         }
117
118         for (Corral c : corrals) {
119             if (c.isValid && c.isPICorral == Corral.POTENTIAL_PI_CORRAL) {
120                 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;
                }
            }
        }
```

```
184
185         return true;
186     }
187
188     protected void mergeWithAdjacents(Board board, Corral corral) {
189         while (corral.adjacentCorrals.size() > 1) {
190             Iterator<Corral> iterator = corral.adjacentCorrals.iterator();
191             Corral adj = null;
192
193             while (iterator.hasNext()) {
194                 adj = iterator.next();
195
196                 if (adj.isPICorral()) {
197                     return;
198                 }
199
200                 if (!adj.containsPlayer) {
201                     break;
202                 }
203             }
204
205             corral = fullyMergeTwoCorrals(board, corral, adj);
206         }
207
208         if (isPICorral(corral)) {
209             corral.isPICorral = Corral.IS_A_PI_CORRAL;
210         } else {
211             corral.isPICorral = Corral.NOT_A_PI_CORRAL;
212         }
213     }
214
215     private Corral fullyMergeTwoCorrals(Board board, Corral a, Corral b) {
216         Corral corral = mergeTwoCorrals(board.getAt(a.getTopX(), a.getTopY()),
          ↪  board.getAt(b.getTopX(), b.getTopY()));
217
218         if (corral == b) {
219             b = a; // this way, we can deal with corral (before a) and b, without doing disjonction.
220         }
221
222         // Merge properties. It is assumed that a and b doesn't contain the player
223         // topX, topY are already updated
224         // the set currentCorrals was also updated.
225         corral.adjacentToPlayerCorral |= b.adjacentToPlayerCorral;
226         corral.onlyCrateOnTarget &= b.onlyCrateOnTarget;
227
228         // update adjacentCorrals
229         // Add all adjacents corral of b to corral, but corral is adjacent to b,
230         // we must remove it. The remove is done before addAll because the resulting
231         // set is likely to be bigger than b one.
232         b.adjacentCorrals.remove(corral);
233         // also update adjacent of b
234         for (Corral bAdj : b.adjacentCorrals) {
235             bAdj.adjacentCorrals.remove(b);
236
237             if (bAdj != corral) {
238                 bAdj.adjacentCorrals.add(corral);
239             }
240         }
241         corral.adjacentCorrals.remove(b);
242         corral.adjacentCorrals.addAll(b.adjacentCorrals);
243
244         // update barrier and crates
245         for (TileInfo tile : b.crates) {
```

```
246             if (!corral.crates.contains(tile)) {
247                 corral.crates.add(tile);
248             }
249         }
250 
251         // merge the two barrier. Some crates aren't in a barrier.
252         for (TileInfo tile : b.barrier) {
253             if (!corral.barrier.contains(tile)) {
254                 corral.barrier.add(tile);
255             }
256         }
257 
258 
259         int[] adjacents = new int[4];
260         int size;
261         for (int i = 0; i < corral.barrier.size(); i++) {
262             TileInfo crate = corral.barrier.get(i);
263             size = 0;
264             for (Direction dir : Direction.VALUES) {
265                 TileInfo tile = crate.adjacent(dir);
266                 if (tile.isSolid()) {
267                     continue;
268                 }
269 
270                 Corral adj = findCorral(tile);
271 
272                 boolean new_ = true;
273                 for (int k = 0; k < size; k++) {
274                     if (adjacents[k] == adj.id) {
275                         new_ = false;
276                         break;
277                     }
278                 }
279 
280                 if (new_) {
281                     adjacents[size] = adj.id;
282                     size++;
283                 }
284             }
285 
286             if (size <= 1) { // not in barrier !
287                 corral.barrier.remove(i);
288                 i--;
289             }
290         }
291 
292         return corral;
293     }
294 
295     /**
296      * Compute adjacent corrals of crates, barriers and various property of Corral
297      */
298     protected void preComputePICorral(Board board, int[] crates) {
299         List<Corral> adj = new ArrayList<>();
300 
301         for (int crateI : crates) {
302             TileInfo crate = board.getAt(crateI);
303 
304             adj.clear();
305 
306             // find adjacent corrals
307             boolean adjacentToPlayerCorral = false;
308             for (Direction dir : Direction.VALUES) {
```

```
309                 TileInfo tile = crate.adjacent(dir);
310                 if (tile.isSolid()) {
311                     continue;
312                 }
313 
314                 Corral corral = findCorral(tile);
315                 // maximal size of adj is 4, so I think that using a list rather than a set is faster
316                 if (!adj.contains(corral)) {
317                     adj.add(corral);
318                 }
319 
320                 if (corral.containsPlayer()) {
321                     adjacentToPlayerCorral = true;
322                 }
323             }
324 
325             if (adj.size() == 1) {
326                 // the crate is inside a corral
327                 // and not a part of a barrier
328                 adj.get(0).crates.add(crate);
329 
330                 if (crate.isCrate()) {
331                     adj.get(0).onlyCrateOnTarget = false;
332                 }
333             } else if (adj.size() > 1) {
334                 // crate is a part of a barrier
335                 for (int i = 0; i < adj.size(); i++) {
336                     Corral corral = adj.get(i);
337                     corral.crates.add(crate);
338                     corral.barrier.add(crate);
339                     corral.adjacentToPlayerCorral |= adjacentToPlayerCorral;
340 
341                     if (crate.isCrate()) {
342                         corral.onlyCrateOnTarget = false;
343                     }
344 
345                     for (int j = i + 1; j < adj.size(); j++) {
346                         Corral corral2 = adj.get(j);
347 
348                         if (corral.adjacentCorrals.add(corral2)) {
349                             corral2.adjacentCorrals.add(corral);
350                         }
351                     }
352                 }
353             }
354         }
355     }
356 
357     /**
358      * Move a node from a aPackage to another. {@code node}
359      * and {@code dest} must be in separate trees.
360      * This method breaks the union find structure.
361      * So, it must be used carefully.
362      */
363     private void addToCorral(TileInfo tile, TileInfo inCorral) {
364         int i = tile.getIndex();
365         int rootI = find(inCorral.getIndex());
366 
367         parent[i] = rootI;
368         rank[i] = 0;
369         rank[rootI] = Math.max(1, rank[rootI]);
370     }
371 
```

```
372     /**
373      * Remove a node from his aPackage and create a new aPackage.
374      * This method breaks the union find structure.
375      * So, it must be used carefully.
376      */
377     private void newCorral(TileInfo tile) {
378         int i = tile.getIndex();
379         parent[i] = i;
380         rank[i] = 0;
381 
382         Corral corral = corrals[i];
383         corral.containsPlayer = false;
384         corral.isPICorral = Corral.POTENTIAL_PI_CORRAL;
385         corral.onlyCrateOnTarget = true;
386         corral.isValid = true;
387         corral.crates.clear();
388         corral.barrier.clear();
389         corral.adjacentCorrals.clear();
390         corral.topX = tile.getX();
391         corral.topY = tile.getY();
392 
393         currentCorrals.add(corral);
394     }
395 
396     private Corral mergeTwoCorrals(TileInfo inCorral1, TileInfo inCorral2) {
397         int corral1I = find(inCorral1.getIndex());
398         int corral2I = find(inCorral2.getIndex());
399 
400         if (corral1I != corral2I) {
401             int oldCorralI;
402             int newCorralI;
403             if (rank[corral1I] < rank[corral2I]) {
404                 oldCorralI = corral1I;
405                 newCorralI = corral2I;
406             } else if (rank[corral1I] > rank[corral2I]) {
407                 oldCorralI = corral2I;
408                 newCorralI = corral1I;
409             } else {
410                 oldCorralI = corral1I;
411                 newCorralI = corral2I;
412                 rank[newCorralI]++;
413             }
414 
415             parent[oldCorralI] = newCorralI;
416 
417             Corral newCorral = corrals[newCorralI];
418             Corral oldCorral = corrals[oldCorralI];
419 
420             oldCorral.isValid = false;
421             currentCorrals.remove(oldCorral);
422             newCorral.containsPlayer |= oldCorral.containsPlayer();
423 
424             if (oldCorral.topY < newCorral.topY || (oldCorral.topY == newCorral.topY && oldCorral.topX
                ↪  < newCorral.topX)) {
425                 newCorral.topX = oldCorral.topX;
426                 newCorral.topY = oldCorral.topY;
427             }
428 
429             return newCorral;
430         }
431 
432         return corrals[corral1I];
433     }
```

```
434
435
436     private int find(int i) {
437         if (parent[i] != i) {
438             int root = find(parent[i]);
439             parent[i] = root;
440
441             return root;
442         }
443
444         return i;
445     }
446
447     /**
448      * The tile must be a non-solid tile: a floor or a target
449      * @param tile a floor or target tile
450      * @return the corral in which the tile is
451      */
452     public Corral findCorral(TileInfo tile) {
453         int i = tile.getIndex();
454
455         if (parent[i] < 0) {
456             return null;
457         }
458
459         return corrals[find(i)];
460     }
461
462     public Collection<Corral> getCorrals() {
463         return currentCorrals;
464     }
465
466     public int getRealNumberOfCorral() {
467         return realNumberOfCorral;
468     }
469
470     public void setDeadlockTable(DeadlockTable table) {
471         for (Corral c : corrals) {
472             c.setDeadlockTable(table);
473         }
474     }
475 }
```

WeightedState

```
1  package fr.valax.sokoshell.solver;
2
3  import fr.valax.sokoshell.utils.SizeOf;
4
5  /**
6   * A simple derivation of State with a weight, i.e. something to rank the states.
7   * Used for instance by {@link AStarSolver}
8   */
9  public class WeightedState extends State {
10
11     private int cost = 0;
12
13     private int heuristic = 0;
14
15     public WeightedState(int playerPos, int[] cratesIndices, int hash, State parent, int cost, int
       ↪  heuristic) {
16         super(playerPos, cratesIndices, hash, parent);
```

```
17          this.setCost(cost);
18          this.setHeuristic(heuristic);
19      }
20
21      public WeightedState(State state, int cost, int heuristic) {
22          this(state.playerPos(), state.cratesIndices(), state.hash(), state.parent(), cost, heuristic);
23      }
24
25      /**
26       * <strong>This function does NOT compute the heuristic of the child state.</strong>
27       * Use {@link WeightedState#setHeuristic(int)} to set it after calling this method.
28       */
29      public WeightedState child(int newPlayerPos, int crateToMove, int crateDestination) {
30          return new WeightedState(super.child(newPlayerPos, crateToMove, crateDestination),
31                  cost(), 0);
32      }
33
34      @Override
35      public long approxSizeOfAccurate() {
36          return SizeOf.getWeightedStateLayout().instanceSize() +
37                  SizeOf.getIntArrayLayout().instanceSize() +
38                  (long) Integer.BYTES * cratesIndices.length;
39      }
40
41      @Override
42      public long approxSizeOf() {
43          return 40 +
44                  16 +
45                  (long) Integer.BYTES * cratesIndices.length;
46      }
47
48      /**
49       * The state weight, which is the sum of its cost and its heuristic.
50       */
51      public int weight() {
52          return cost() + heuristic();
53      }
54
55      /**
56       * The cost the come to this state.
57       */
58      public int cost() {
59          return cost;
60      }
61
62      public void setCost(int cost) {
63          this.cost = cost;
64      }
65
66      /**
67       * The heuristic between this state and a solution.
68       */
69      public int heuristic() {
70          return heuristic;
71      }
72
73      public void setHeuristic(int heuristic) {
74          this.heuristic = heuristic;
75      }
76  }
```

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();
```

```
63              solution.writeSolution(jpw);
64              jpw.endObject();
65          }
66      }
67
68      /**
69       * Returns the player position on the x-axis at the beginning
70       *
71       * @return the player position on the x-axis at the beginning
72       */
73      public int getPlayerX() {
74          return playerPos % getWidth();
75      }
76
77      /**
78       * Returns the player position on the y-axis at the beginning
79       *
80       * @return the player position on the y-axis at the beginning
81       */
82      public int getPlayerY() {
83          return playerPos / getWidth();
84      }
85
86      /**
87       * Returns the initial state i.e. a state representing the level at the beginning
88       *
89       * @return the initial state
90       */
91      public State getInitialState() {
92          State.initZobristValues(getWidth() * getHeight()); // TODO
93
94          List<Integer> cratesIndices = new ArrayList<>();
95
96          for (int y = 0; y < getHeight(); y++) {
97              for (int x = 0; x < getWidth(); x++) {
98                  if (getAt(x, y).anyCrate()) {
99                      cratesIndices.add(y * getWidth() + x);
100                 }
101             }
102         }
103
104         int[] cratesIndicesArray = new int[cratesIndices.size()];
105         for (int i = 0; i < cratesIndices.size(); i++) {
106             cratesIndicesArray[i] = cratesIndices.get(i);
107         }
108
109         return new State(playerPos, cratesIndicesArray, null);
110     }
111
112     public BigInteger estimateNumberOfState() {
113         if (maxNumberOfStateEstimation == null) {
114             // + 1 for numberOfCrate because we also consider the player
115             maxNumberOfStateEstimation = Utils.binomial(numberOfNonWalls, numberOfCrates + 1);
116         }
117
118         return maxNumberOfStateEstimation;
119     }
120
121     public BigInteger estimateNumberOfState(int nDeadTile) {
122         int nFloor = numberOfNonWalls - nDeadTile;
123
124         return Utils.binomial(nFloor, numberOfCrates + 1);
125     }
```

```
126
127     /**
128      * @return the number of crate in this level
129      */
130     public int getNumberOfCrates() {
131         return numberOfCrates;
132     }
133
134     /**
135      * @return the number of non-wall (floor, target, crate, crate on target)
136      */
137     public int getNumberOfNonWalls() {
138         return numberOfNonWalls;
139     }
140
141     /**
142      * Returns the last solver report that is a solution
143      * @return the last solver report that is a solution
144      */
145     public SolverReport getLastSolution() {
146         if (solverReports.isEmpty()) {
147             return null;
148         }
149
150         for (int i = solverReports.size() - 1; i >= 0; i--) {
151             SolverReport r = solverReports.get(i);
152
153             if (r.isSolved()) {
154                 return r;
155             }
156         }
157
158         return null;
159     }
160
161     /**
162      * Returns the last report
163      *
164      * @return the last report
165      */
166     public SolverReport getLastReport() {
167         if (solverReports.isEmpty()) {
168             return null;
169         } else {
170             return solverReports.get(solverReports.size() - 1);
171         }
172     }
173
174     /**
175      * Returns the solver report at the specified position
176      *
177      * @param index index of the report to return
178      * @return the solver report at the specified position
179      */
180     public SolverReport getSolverReport(int index) {
181         if (index < 0 || index >= solverReports.size()) {
182             return null;
183         } else {
184             return solverReports.get(index);
185         }
186     }
187
188     /**
```

```
189      * Returns all solver reports
190      *
191      * @return all solver reports
192      */
193     public List<SolverReport> getSolverReports() {
194         return solverReports;
195     }
196 
197     /**
198      * Returns the number of solver report
199      *
200      * @return the number of solver report
201      */
202     public int numberOfSolverReport() {
203         return solverReports.size();
204     }
205 
206     /**
207      * Add a solver report to this level
208      *
209      * @param solverReport the report to add
210      * @throws IllegalArgumentException if the report isn't for this level
211      */
212     public synchronized void addSolverReport(SolverReport solverReport) {
213         if (solverReport.getParameters().getLevel() != this) {
214             throw new IllegalArgumentException("Attempting to add a report to the wrong level");
215         }
216         solverReports.add(solverReport);
217     }
218 
219     public synchronized void removeSolverReport(int index) {
220         solverReports.remove(index);
221     }
222 
223     public synchronized int indexOf(SolverReport solverReport) {
224         if (solverReport.getParameters().getLevel() != this) {
225             return -1;
226         }
227         return solverReports.indexOf(solverReport);
228     }
229 
230     /**
231      * Returns if an attempt to solve this level was done. It doesn't mean that this level has a
    ↪ solution
232      *
233      * @return {@code true} if an attempt to solve this level was done.
234      */
235     public boolean hasReport() {
236         return solverReports.size() > 0;
237     }
238 
239     /**
240      * Returns {@code true} if this level has a solution
241      *
242      * @return {@code true} if this level has a solution
243      */
244     public boolean hasSolution() {
245         for (int i = 0; i < solverReports.size(); i++) {
246             SolverReport r = solverReports.get(i);
247             if (r.isSolved()) {
248                 return true;
249             }
250         }
```

```
251
252         return false;
253     }
254
255     /**
256      * Returns the index of this level in the pack
257      *
258      * @return the index of this level in the pack
259      */
260     public int getIndex() {
261         return index;
262     }
263
264     /**
265      * Returns the pack in which this level is
266      *
267      * @return the pack in which this level is
268      */
269     public Pack getPack() {
270         return pack;
271     }
272
273
274     /**
275      * A builder of {@link Level}
276      */
277     public static class Builder {
278
279         private int playerX = -1;
280         private int playerY = -1;
281
282         private Tile[][] board = new Tile[0][0];
283         private int width;
284         private int height;
285         private int index;
286
287         /**
288          * Builds and returns a {@link Level}
289          *
290          * @return the new {@link Level}
291          * @throws BuilderException if the player is outside the board
292          * r the player is on a solid tile
293          */
294         public Level build() {
295             if (board == null) {
296                 throw new BuilderException("Board is null");
297             }
298
299             if (playerX < 0 || playerX >= width) {
300                 throw new BuilderException("Player x out of bounds");
301             }
302
303             if (playerY < 0 || playerY >= height) {
304                 throw new BuilderException("Player y out of bounds");
305             }
306
307             if (board[playerY][playerX].isSolid()) {
308                 throw new BuilderException("Player is on a solid tile");
309             }
310
311             formatLevel();
312
313             return new Level(board, width, height, playerY * width + playerX, index);
```

```
314         }
315
316         /**
317          * Format the level for the solver. Some levels aren't surrounded by wall
318          * or have rooms that are inaccessible. This method removes these rooms
319          * and add wall if necessary.
320          */
321         private void formatLevel() {
322             Set<Integer> visited = new HashSet<>();
323
324             int i = 0;
325             for (int y = 0; y < height; y++) {
326                 for (int x = 0; x < width; x++) {
327                     if (board[y][x] != Tile.WALL && !visited.contains(i)) {
328                         addWallIfNecessary(x, y, visited);
329                     }
330
331                     i++;
332                 }
333             }
334
335             surroundByWallIfNecessary();
336         }
337
338         private void addWallIfNecessary(int x, int y, Set<Integer> visited) {
339             boolean needWall = true;
340
341             Set<Integer> localVisited = new HashSet<>();
342             Stack<Integer> toVisit = new Stack<>();
343             toVisit.add(y * width + x);
344             localVisited.add(toVisit.peek());
345
346             while (!toVisit.isEmpty()) {
347                 int i = toVisit.pop();
348
349                 int x2 = i % width;
350                 int y2 = i / width;
351
352                 if (x2 == playerX && y2 == playerY) {
353                     needWall = false;
354                 }
355
356                 for (Direction d : Direction.VALUES) {
357                     int x3 = x2 + d.dirX();
358                     int y3 = y2 + d.dirY();
359
360                     if (x3 < 0 || x3 >= width || y3 < 0 || y3 >= height) {
361                         continue;
362                     }
363
364                     int i3 = y3 * width + x3;
365
366                     if (board[y3][x3] != Tile.WALL && localVisited.add(i3)) {
367                         visited.add(i3);
368                         toVisit.push(i3);
369                     }
370                 }
371             }
372
373             if (needWall) {
374                 for (Integer i : localVisited) {
375                     int x2 = i % width;
376                     int y2 = i / width;
```

```
377
378                 board[y2][x2] = Tile.WALL;
379             }
380         }
381     }
382
383     private void surroundByWallIfNecessary() {
384         int left = 0;
385         int right = 0;
386         int top = 0;
387         int bottom = 0;
388
389         for (int y = 0; y < height; y++) {
390             if (board[y][0] != Tile.WALL) {
391                 left = 1;
392             }
393             if (board[y][width - 1] != Tile.WALL) {
394                 right = 1;
395             }
396         }
397
398         for (int x = 0; x < width; x++) {
399             if (board[0][x] != Tile.WALL) {
400                 top = 1;
401             }
402             if (board[height - 1][x] != Tile.WALL) {
403                 bottom = 1;
404             }
405         }
406
407         if (left == 0 && right == 0 && top == 0 && bottom == 0) {
408             return;
409         }
410
411         Tile[][] newTiles = new Tile[height + top + bottom][width + right + left];
412
413         for (int y = 0; y < height + top + bottom; y++) {
414             for (int x = 0; x < width + right + left; x++) {
415                 if (x >= left && y >= top && x < width + left && y < height + top) {
416                     newTiles[y][x] = board[y - top][x - left];
417                 } else {
418                     newTiles[y][x] = Tile.WALL;
419                 }
420             }
421         }
422
423         board = newTiles;
424         width += right + left;
425         height += top + bottom;
426         playerX += left;
427         playerY += top;
428     }
429
430     /**
431      * Returns the player position on the x-axis
432      *
433      * @return the player position on the x-axis
434      */
435     public int getPlayerX() {
436         return playerX;
437     }
438
439     /**
```

```
440         * Returns the player position on the y-axis
441         *
442         * @return the player position on the y-axis
443         */
444        public int getPlayerY() {
445            return playerY;
446        }
447
448        /**
449         * Set the player position to (x, y)
450         *
451         * @param x player position on the x-axis
452         * @param y player position on the y-axis
453         */
454        public void setPlayerPos(int x, int y) {
455            this.playerX = x;
456            this.playerY = y;
457        }
458
459        /**
460         * Set the player position on the x-axis to x
461         *
462         * @param playerX the new player position on the x-axis
463         */
464        public void setPlayerX(int playerX) {
465            this.playerX = playerX;
466        }
467
468        /**
469         * Set the player position on the y-axis to x
470         *
471         * @param playerY the new player position on the y-axis
472         */
473        public void setPlayerY(int playerY) {
474            this.playerY = playerY;
475        }
476
477        private void resizeIfNeeded(int minWidth, int minHeight) {
478            setSize(Math.max(minWidth, width),
479                    Math.max(minHeight, height));
480        }
481
482        /**
483         * Resize this level to (newWidth, newHeight). If dimensions are higher than the old one,
484         * new tiles are filled with WALL. For other, tiles are the same.
485         *
486         * @param newWidth the new width of the level
487         * @param newHeight the new width of the level
488         */
489        public void setSize(int newWidth, int newHeight) {
490            if (newWidth == width && newHeight == height) {
491                return;
492            }
493
494            Tile[][] newBoard = new Tile[newHeight][newWidth];
495
496            int yMax = Math.min(newHeight, height);
497            int xMax = Math.min(newWidth, width);
498            for (int y = 0; y < yMax; y++) {
499                System.arraycopy(board[y], 0, newBoard[y], 0, xMax);
500
501                for (int x = xMax; x < newWidth; x++) {
502                    newBoard[y][x] = Tile.WALL;
```

```
503                 }
504             }
505 
506             board = newBoard;
507 
508             width = newWidth;
509             height = newHeight;
510         }
511 
512         /**
513          * Returns the width of the level
514          *
515          * @return the width of the level
516          */
517         public int getWidth() {
518             return width;
519         }
520 
521         /**
522          * Sets the width of the level
523          *
524          * @param width the new width of the level
525          * @see #setSize(int, int)
526          */
527         public void setWidth(int width) {
528             setSize(width, height);
529         }
530 
531         /**
532          * Returns the height of the level
533          *
534          * @return the height of the level
535          */
536         public int getHeight() {
537             return height;
538         }
539 
540         /**
541          * Sets the height of the level
542          *
543          * @param height the new height of the level
544          * @see #setSize(int, int)
545          */
546         public void setHeight(int height) {
547             setSize(width, height);
548         }
549 
550         /**
551          * Set at (x, y) the tile. If (x, y) is outside the level, the level is resized
552          *
553          * @param tile the new tile
554          * @param x x position
555          * @param y y position
556          */
557         public void set(Tile tile, int x, int y) {
558             resizeIfNeeded(x, y);
559             board[y][x] = tile;
560         }
561 
562         /**
563          * Returns the tile at (x, y)
564          * @param x x position of the tile
565          * @param y y position of the tile
```

```
566          * @return the tile at (x, y)
567          */
568         public Tile get(int x, int y) {
569             if (x < 0 || x >= width || y < 0 || y >= height) {
570                 return null;
571             }
572 
573             return board[y][x];
574         }
575 
576         /**
577          * Returns the index of the level
578          * @return the index of the level
579          */
580         public int getIndex() {
581             return index;
582         }
583 
584         /**
585          * Sets the index of the level
586          * @param index the new index of the level
587          */
588         public void setIndex(int index) {
589             this.index = index;
590         }
591     }
592 }
```

SolverReport

```
1  package fr.valax.sokoshell.solver;
2  
3  import fr.poulpogaz.json.JsonException;
4  import fr.poulpogaz.json.JsonPrettyWriter;
5  import fr.poulpogaz.json.JsonReader;
6  import fr.valax.sokoshell.SokoShell;
7  import fr.valax.sokoshell.graphics.style.BoardStyle;
8  import fr.valax.sokoshell.solver.board.Board;
9  import fr.valax.sokoshell.solver.board.Move;
10 import fr.valax.sokoshell.solver.board.MutableBoard;
11 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
12 import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
13 import fr.valax.sokoshell.solver.pathfinder.Node;
14 
15 import java.io.*;
16 import java.util.*;
17 import java.util.stream.Collectors;
18 
19 /**
20  * An object representing the output of a solver. It contains the parameters given to the solver,
21  * some statistics, the solver status and if the status is {@link SolverReport#SOLUTION_FOUND},
22  * it contains two representation of the solution: a sequence of {@link State} and a sequence of
    ↪  {@link Move}.
23  *
24  * @see SolverParameters
25  * @see ISolverStatistics
26  * @see State
27  * @see Move
28  * @author PoulpoGaz
29  * @author darth-mole
30  */
31 public class SolverReport {
```

```
32
33      public static final String NO_SOLUTION = "No solution";
34      public static final String SOLUTION_FOUND = "Solution found";
35      public static final String STOPPED = "Stopped";
36      public static final String TIMEOUT = "Timeout";
37      public static final String RAM_EXCEED = "Ram exceed";
38
39      /**
40       * Creates and returns a report that doesn't contain a solution
41       *
42       * @param params the parameters of the solver
43       * @param stats the statistics
44       * @param status the solver status
45       * @return a report without a solution
46       * @throws IllegalArgumentException if the state is {@link SolverReport#SOLUTION_FOUND}
47       */
48      public static SolverReport withoutSolution(SolverParameters params, ISolverStatistics stats,
        ↪  String status) {
49          return new SolverReport(params, stats, null, status);
50      }
51
52      /**
53       * Creates and returns a report containing a solution. The solution is determined
54       * from the final state.
55       *
56       * @param finalState the final state
57       * @param params the parameters of the solver
58       * @param stats the statistics
59       * @return a report with a solution
60       */
61      public static SolverReport withSolution(State finalState, SolverParameters params,
        ↪  ISolverStatistics stats) {
62          List<State> solution = new ArrayList<>();
63
64          State s = finalState;
65          while (s.parent() != null)
66          {
67              solution.add(s);
68              s = s.parent();
69          }
70          solution.add(s);
71          Collections.reverse(solution);
72
73          return new SolverReport(params, stats, solution, SOLUTION_FOUND);
74      }
75
76      private final SolverParameters parameters;
77      private final ISolverStatistics statistics;
78
79      private final String status;
80
81      /**
82       * Solution packed in an int array.
83       * Three bits are used for storing a move.
84       * Move 1 is located at bit 0 of array 0,
85       * Move 2 is located at bit 3 of array 0,
86       * ...,
87       * Move 10 is located at bit 27 of array 0,
88       * Move 11 is located at bit 30 of array 0
89       * and use the first bit of array 1.
90       * Move 12 is located at bit 1 of array 1,
91       * etc.
92       * Bits are stored in little-endian fashion.
```

```
93       */
94      private final int[] solution;
95      private final int numberOfMoves;
96      private final int numberOfPushes;
97
98      public SolverReport(SolverParameters parameters,
99                          ISolverStatistics statistics,
100                         List<State> states,
101                         String status) {
102         this.parameters = Objects.requireNonNull(parameters);
103         this.statistics = Objects.requireNonNull(statistics);
104         this.status = Objects.requireNonNull(status);
105
106         if (status.equals(SOLUTION_FOUND)) {
107             if (states == null) {
108                 throw new
                    ↪  IllegalArgumentException("SolverStatus is SOLUTION_FOUND. You must give the solution");
109             }
110
111             SolutionBuilder builder = createFullSolution(states);
112
113             numberOfPushes = builder.getNumberOfPushes();
114             numberOfMoves = builder.getNumberOfMoves();
115             solution = builder.getSolution();
116         } else {
117             numberOfMoves = -1;
118             numberOfPushes = -1;
119             solution = null;
120         }
121     }
122
123     private SolverReport(SolverParameters parameters,
124                          ISolverStatistics statistics,
125                          String status,
126                          SolutionBuilder builder) {
127         this.parameters = Objects.requireNonNull(parameters);
128         this.statistics = Objects.requireNonNull(statistics);
129         this.status = Objects.requireNonNull(status);
130
131         if (status.equals(SOLUTION_FOUND)) {
132             numberOfPushes = builder.getNumberOfPushes();
133             numberOfMoves = builder.getNumberOfMoves();
134             solution = builder.getSolution();
135         } else {
136             numberOfMoves = -1;
137             numberOfPushes = -1;
138             solution = null;
139         }
140     }
141
142
143     /**
144      * Deduce from solution's states all the moves needed to solve the sokoban
145      *
146      * @return the full solution
147      */
148     private SolutionBuilder createFullSolution(List<State> states) {
149         Level level = parameters.getLevel();
150         Board board = new MutableBoard(level);
151
152         SolutionBuilder sb = new SolutionBuilder(2 * states.size());
153         List<Move> temp = new ArrayList<>();
154
```

```
155         TileInfo player = board.getAt(level.getPlayerX(), level.getPlayerY());
156
157         CrateAStar aStar = new CrateAStar(board);
158         for (int i = 0; i < states.size() - 1; i++) {
159             State current = states.get(i);
160
161             if (i != 0) {
162                 board.addStateCrates(current);
163             }
164
165             State next = states.get(i + 1);
166             StateDiff diff = getStateDiff(board, current, next);
167
168             Node node = aStar.findPathAndComputeMoves(
169                     player, null,
170                     diff.crate(), diff.crateDest());
171
172             if (node == null) {
173                 throw cannotFindPathException(board, current, next);
174             }
175
176             player = node.getPlayer();
177             while (node.getParent() != null) {
178                 temp.add(node.getMove());
179                 node = node.getParent();
180             }
181
182             sb.ensureCapacity(sb.getNumberOfMoves() + temp.size());
183             for (int j = temp.size() - 1; j >= 0; j--) {
184                 sb.add(temp.get(j));
185             }
186             temp.clear();
187
188             board.removeStateCrates(current);
189         }
190
191         return sb;
192     }
193
194     /**
195      * Find the differences between two states:
196      * <ul>
197      *     <li>new player position</li>
198      *     <li>old crate pos</li>
199      *     <li>new crate pos</li>
200      * </ul>
201      *
202      * @param board the board
203      * @param from the first state
204      * @param to the second state
205      * @return a {@link StateDiff}
206      */
207     private StateDiff getStateDiff(Board board, State from, State to) {
208         List<Integer> state1Crates =
             ↪  Arrays.stream(from.cratesIndices()).boxed().collect(Collectors.toList());
209         List<Integer> state2Crates =
             ↪  Arrays.stream(to.cratesIndices()).boxed().collect(Collectors.toList());
210
211         List<Integer> state1Copy = state1Crates.stream().toList();
212         state1Crates.removeAll(state2Crates);
213         state2Crates.removeAll(state1Copy);
214
215         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();

```

```
277         jpw.value(Base64.getEncoder().encodeToString(baos.toByteArray()));
278     }
279
280
281     public static SolverReport fromJson(JsonReader jr, Level level) throws JsonException, IOException
    ↪   {
282         String status = jr.assertKeyEquals("status").nextString();
283
284         jr.assertKeyEquals("parameters");
285         SolverParameters parameters = SolverParameters.fromJson(jr, level);
286
287         String key = jr.nextKey();
288
289         SolutionBuilder sb = null;
290         if (key.equals("solution")) {
291             jr.beginArray();
292
293             sb = new SolutionBuilder(32 * 5); // uses array of size 16
294             while (!jr.isArrayEnd()) {
295                 String name = jr.nextString();
296                 Move move = Move.of(name);
297
298                 if (move == null) {
299                     throw new IOException("Unknown move: " + name);
300                 }
301
302                 sb.add(move);
303             }
304             jr.endArray();
305
306             jr.assertKeyEquals("statistics");
307         } else if (!key.equals("statistics")) {
308             throw new
                ↪   JsonException(String.format("Invalid key. Expected \"statistics\" but was \"%s\"",
                ↪   key));
309         }
310
311         // see writeSolution
312         byte[] bytes = Base64.getDecoder().decode(jr.nextString());
313
314         ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));
315         ISolverStatistics stats;
316         try {
317             stats = (ISolverStatistics) ois.readObject();
318         } catch (ClassNotFoundException e) {
319             throw new IOException(e);
320         }
321         ois.close();
322
323         return new SolverReport(parameters, stats, status, sb);
324     }
325
326
327     /**
328      * Returns the type of the solver used to produce this report
329      *
330      * @return the type of the solver used to produce this report
331      */
332     public String getSolverName() {
333         return parameters.getSolverName();
334     }
335
336     /**
```

```
337      * Returns the parameters given to the solver that produce this report
338      *
339      * @return the parameters given to the solver
340      */
341     public SolverParameters getParameters() {
342         return parameters;
343     }
344 
345     /**
346      * Returns the statistics produce by the solver that produce this report.
347      * However, {@linkplain Solver solvers} are only capable of recording when
348      * the research start and end. Others statistics are produced by {@link Tracker}
349      *
350      * @return the parameters given to the solver
351      */
352     public ISolverStatistics getStatistics() {
353         return statistics;
354     }
355 
356     public SolutionIterator getSolutionIterator() {
357         if (solution == null) {
358             return null;
359         }
360 
361         return new SolutionIterator();
362     }
363 
364     /**
365      * If the sokoban was solved, this report contains the solution as a sequence
366      * of moves. It describes all moves made by the player.
367      *
368      * @return the solution or {@code null} if the sokoban wasn't solved
369      */
370     public List<Move> getFullSolution() {
371         if (solution == null) {
372             return null;
373         }
374 
375         ListIterator<Move> it = getSolutionIterator();
376         List<Move> moves = new ArrayList<>(numberOfMoves);
377 
378         while (it.hasNext()) {
379             moves.add(it.next());
380         }
381 
382         return moves;
383     }
384 
385     /**
386      * Returns the number of pushes the player made to solve the sokoban
387      *
388      * @return {@code -1} if the sokoban wasn't solved or the number of pushes the player made to
    ↪  solve the sokoban
389      */
390     public int numberOfPushes() {
391         return numberOfPushes;
392     }
393 
394     /**
395      * Returns the number of moves the player made to solve the sokoban
396      *
397      * @return {@code -1} if the sokoban wasn't solved or the number of moves the player made to solve
    ↪  the sokoban
```

```
398      */
399     public int numberOfMoves() {
400         return numberOfMoves;
401     }
402
403
404     /**
405      * Returns {@code true} if this report contains a solution
406      *
407      * @return {@code true} if this report contains a solution
408      */
409     public boolean isSolved() {
410         return status.equals(SOLUTION_FOUND);
411     }
412
413     /**
414      * Returns {@code true} if this report doesn't contain a solution
415      *
416      * @return {@code true} if this report doesn't contain a solution
417      */
418     public boolean hasNoSolution() {
419         return !status.equals(SOLUTION_FOUND);
420     }
421
422     /**
423      * Returns {@code true} if the solver was stopped by the user
424      *
425      * @return {@code true} if the solver was stopped by the user
426      */
427     public boolean isStopped() {
428         return status.equals(STOPPED);
429     }
430
431
432     public String getStatus() {
433         return status;
434     }
435
436     /**
437      * Returns the level that was given to the solver
438      *
439      * @return the level that was given to the solver
440      */
441     public Level getLevel() {
442         return parameters.getLevel();
443     }
444
445
446     /**
447      * Returns the pack of the level that was given to the solver
448      *
449      * @return the pack of the level that was given to the solver
450      */
451     public Pack getPack() {
452         return parameters.getLevel().getPack();
453     }
454
455     /**
456      * Contains all differences between two states except the old player position.
457      *
458      * @param playerDest player destination
459      * @param crate old crate position
460      * @param crateDest crate destination
```

```
461      */
462     private record StateDiff(TileInfo playerDest, TileInfo crate, TileInfo crateDest) {}
463
464     /**
465      * An object to iterate over a solution in forward and backward order.
466      */
467     public class SolutionIterator implements ListIterator<Move> {
468
469         /**
470          * Position in the array
471          */
472         private int arrayPos;
473
474         /**
475          * Position in solution[arrayPos]
476          */
477         private int bitPos;
478
479         private int move;
480         private int push;
481
482         /**
483          * @return read the next bit
484          */
485         private int readNext() {
486             int bit = (solution[arrayPos] >> bitPos) & 0b1;
487
488             bitPos++;
489             if (bitPos == 32) {
490                 bitPos = 0;
491                 arrayPos++;
492             }
493
494             return bit;
495         }
496
497         /**
498          * @return read the previous bit
499          */
500         private int readPrevious() {
501             bitPos--;
502             if (bitPos < 0) {
503                 bitPos = 31;
504                 arrayPos--;
505             }
506
507             return (solution[arrayPos] >> bitPos) & 0b1;
508         }
509
510
511         @Override
512         public boolean hasNext() {
513             return move < numberOfMoves;
514         }
515
516         @Override
517         public Move next() {
518             if (!hasNext()) {
519                 throw new NoSuchElementException();
520             }
521
522             int first  = readNext();
523             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
        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
```

```
587         public void set(Move move) {
588             throw new UnsupportedOperationException();
589         }
590 
591         @Override
592         public void add(Move move) {
593             throw new UnsupportedOperationException();
594         }
595 
596         public int getMoveCount() {
597             return move;
598         }
599 
600         public int getPushCount() {
601             return push;
602         }
603     }
604 
605     /**
606      * A convenience object to convert a list of move to a solution array.
607      */
608     private static class SolutionBuilder {
609 
610         private int[] solution;
611 
612         private int arrayPos;
613         private int bitPos;
614 
615         private int numberOfMoves;
616         private int numberOfPushes;
617 
618         public SolutionBuilder(int estimatedNumberOfMove) {
619             solution = new int[computeArraySize(estimatedNumberOfMove)];
620         }
621 
622         private void write(int bit) {
623             solution[arrayPos] = (bit & 0b1) << bitPos | solution[arrayPos];
624 
625             bitPos++;
626             if (bitPos == 32) {
627                 bitPos = 0;
628                 arrayPos++;
629             }
630         }
631 
632         public void add(Move move) {
633             if (bitPos + 3 >= 32 && arrayPos + 1 >= solution.length) {
634                 ensureCapacity(numberOfMoves * 2 + 1);
635             }
636 
637             int value = move.ordinal();
638             write(value & 0b1);
639             write((value >> 1) & 0b1);
640             write((value >> 2) & 0b1);
641             numberOfMoves++;
642 
643             if (move.moveCrate()) {
644                 numberOfPushes++;
645             }
646         }
647 
648         public void ensureCapacity(int numberOfMove) {
649             int minArraySize = computeArraySize(numberOfMove);
```

```
650
651             if (minArraySize > solution.length) {
652                 solution = Arrays.copyOf(solution, minArraySize);
653             }
654         }
655
656         public int getNumberOfMoves() {
657             return numberOfMoves;
658         }
659
660         public int getNumberOfPushes() {
661             return numberOfPushes;
662         }
663
664         public int[] getSolution() {
665             int arraySize = computeArraySize(numberOfMoves);
666
667             return Arrays.copyOf(solution, arraySize);
668         }
669
670         private int computeArraySize(int numberOfMove) {
671             int nBits = 3 * numberOfMove;
672
673             return nBits / 32 + 1;
674         }
675     }
676 }
```

FESS0Solver

```
1  package fr.valax.sokoshell.solver;
2
3  import fr.valax.sokoshell.solver.board.Direction;
4  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
5  import fr.valax.sokoshell.solver.collections.SolverCollection;
6  import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
7  import fr.valax.sokoshell.solver.heuristic.Heuristic;
8
9  import java.util.PriorityQueue;
10
11 public class FESS0Solver extends AbstractSolver<FESS0Solver.FESS0State> {
12
13     private Heuristic heuristic;
14     private int lowerBound;
15
16     public FESS0Solver() {
17         super("fess0");
18     }
19
20     @Override
21     protected void init(SolverParameters parameters) {
22         heuristic = new GreedyHeuristic(board);
23         toProcess = new SolverPriorityQueue();
24     }
25
26     @Override
27     protected void addInitialState(Level level) {
28         CorralDetector detector = board.getCorralDetector();
29         State s = level.getInitialState();
30
31         board.addStateCrates(s);
```

```
32          detector.findCorral(board, s.playerPos() % level.getWidth(), s.playerPos() /
            ↪  level.getWidth());
33          board.removeStateCrates(s);
34
35          lowerBound = heuristic.compute(s);
36
37          FESS0State state = new FESS0State(s, 0, lowerBound, detector.getRealNumberOfCorral(),
            ↪  countPackedCrate(s));
38
39          toProcess.addState(state);
40      }
41
42      @Override
43      protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
44          if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
45              return;
46          }
47
48          final int i = board.topLeftReachablePosition(crate, crateDest);
49          // The new player position is the crate position
50          FESS0State s = toProcess.cachedState().child(i, crate.getCrateIndex(), crateDest.getIndex());
51          s.setHeuristic(heuristic.compute(s));
52          s.setConnectivity(board.getCorralDetector().getRealNumberOfCorral());
53          s.setPacking(countPackedCrate(s));
54
55          if (processed.add(s)) {
56              toProcess.addState(s);
57          }
58      }
59
60      private int countPackedCrate(State state) {
61          int nPacked = 0;
62          for (int crate : state.cratesIndices()) {
63              TileInfo tile = board.getAt(crate);
64              if (tile.isTarget() || tile.isCrateOnTarget()) {
65                  nPacked++;
66              }
67          }
68
69          return nPacked;
70      }
71
72      @Override
73      public int lowerBound() {
74          return lowerBound;
75      }
76
77      private static class SolverPriorityQueue extends PriorityQueue<FESS0State>
78              implements SolverCollection<FESS0State> {
79
80          private FESS0State cachedState;
81
82          @Override
83          public void addState(FESS0State state) {
84              offer(state);
85          }
86
87          @Override
88          public FESS0State popState() {
89              return poll();
90          }
91
92          @Override
```

```
93          public FESS0State peekState() {
94              return peek();
95          }
96
97          @Override
98          public FESS0State peekAndCacheState() {
99              cachedState = popState();
100             return cachedState;
101         }
102
103         @Override
104         public FESS0State cachedState() {
105             return cachedState;
106         }
107     }
108
109     protected static class FESS0State extends WeightedState implements Comparable<FESS0State> {
110
111         private int connectivity;
112         private int packing;
113
114         public FESS0State(int playerPos, int[] cratesIndices, int hash, State parent, int cost, int
        ↪  heuristic) {
115             super(playerPos, cratesIndices, hash, parent, cost, heuristic);
116         }
117
118         public FESS0State(State state, int cost, int heuristic, int connectivity, int packing) {
119             super(state, cost, heuristic);
120             this.connectivity = connectivity;
121             this.packing = packing;
122         }
123
124         @Override
125         public FESS0State child(int newPlayerPos, int crateToMove, int crateDestination) {
126             return new FESS0State(super.child(newPlayerPos, crateToMove, crateDestination),
127                     cost(), 0, 0, 0);
128         }
129
130         public int getConnectivity() {
131             return connectivity;
132         }
133
134         public void setConnectivity(int connectivity) {
135             this.connectivity = connectivity;
136         }
137
138         public int getPacking() {
139             return packing;
140         }
141
142         public void setPacking(int packing) {
143             this.packing = packing;
144         }
145
146         @Override
147         public int compareTo(FESS0State o) {
148             // compare in reverse order because
149             // java PriorityQueue is a min-queue
150             return compare(o, this);
151         }
152
153         private static int compare(FESS0State a, FESS0State b) {
154             // -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

```
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
```

```
42      * @return the parameter named param
43      */
44     public SolverParameter get(String param) {
45         return parameters.get(param);
46     }
47 
48     /**
49      *
50      * @param param name of the parameter
51      * @return argument of parameter param or default value
52      * @param <T> type of the argument
53      * @throws ClassCastException if the argument can't be cast to a T
54      */
55     @SuppressWarnings("unchecked")
56     public <T> T getArgument(String param) {
57         SolverParameter p = parameters.get(param);
58 
59         if (p == null) {
60             throw new NoSuchElementException("No such parameter: " + param);
61         }
62 
63         return (T) p.getOrDefault();
64     }
65 
66     /**
67      * @return all parameters
68      */
69     public Collection<SolverParameter> getParameters() {
70         return parameters.values();
71     }
72 
73     /**
74      * @return the level to solve
75      */
76     public Level getLevel() {
77         return level;
78     }
79 
80     /**
81      * @return the name of the solver used
82      */
83     public String getSolverName() {
84         return solverName;
85     }
86 
87 
88     public void append(IJsonWriter jw) throws JsonException, IOException {
89         jw.beginObject();
90         jw.field("solver", solverName);
91 
92         for (Map.Entry<String, SolverParameter> param : parameters.entrySet()) {
93             if (param.getValue().hasArgument()) {
94                 jw.key(param.getKey());
95                 param.getValue().toJson(jw);
96             }
97         }
98 
99         jw.endObject();
100    }
101 
102    public static SolverParameters fromJson(IJsonReader jr, Level level) throws JsonException,
       ↪  IOException {
103        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

```
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);
    }

```

```
36      private static boolean checkFreezeDeadlockRec(TileInfo current, Direction axis) {
37          boolean deadlock = false;
38
39          TileInfo left = current.adjacent(axis);
40          TileInfo right = current.adjacent(axis.negate());
41
42          if (left.isWall() || right.isWall()) { // rule 1
43              deadlock = true;
44
45          } else if (left.isDeadTile() && right.isDeadTile()) { // rule 2
46              deadlock = true;
47
48          } else { // rule 3
49              Tile oldCurr = current.getTile();
50              current.setTile(Tile.WALL);
51
52              if (left.anyCrate()) {
53                  deadlock = checkFreezeDeadlockRec(left);
54              }
55
56              if (!deadlock && right.anyCrate()) {
57                  deadlock = checkFreezeDeadlockRec(right);
58              }
59
60              current.setTile(oldCurr);
61          }
62
63          return deadlock;
64      }
65  }
```

ISolverStatistics

```
1   package fr.valax.sokoshell.solver;
2
3   import fr.valax.sokoshell.utils.PrettyTable;
4   import fr.valax.sokoshell.utils.Utils;
5
6   import java.io.PrintStream;
7   import java.io.Serializable;
8
9   /**
10   * An object that contains various statistics about a solution, including
11   * time start and end, number of node explored and queue size at a specific instant
12   */
13  public interface ISolverStatistics extends Serializable {
14
15      /**
16       * Returns the time in millis when the solver was started
17       *
18       * @return the time in millis when the solver was started
19       */
20      long timeStarted();
21
22      /**
23       * Returns the time in millis when the solver stopped running
24       *
25       * @return the time in millis when the solver stopped running
26       */
27      long timeEnded();
28
29      /**
```

```
30      * Returns the time used by the solver to solve a level
31      *
32      * @return the run time in millis
33      */
34     default long runTime() {
35         return timeEnded() - timeStarted();
36     }
37
38     /**
39      * Returns the total number of state explored by the solver.
40      * If the solver doesn't use State or the {@link Tracker}
41      * doesn't compute this property, implementations can return
42      * a negative number
43      *
44      * @return total number of state explored
45      */
46     int totalStateExplored();
47
48     /**
49      * @return number of state explored per seconds or -1
50      */
51     long stateExploredPerSeconds();
52
53     /**
54      * @return average queue size or -1
55      */
56     int averageQueueSize();
57
58     /**
59      * @return lower bound or -1
60      */
61     int lowerBound();
62
63     /**
64      * Print statistics to out.
65      *
66      * @param out standard output stream
67      * @param err error output stream
68      * @return an optional table containing statistics
69      */
70     default PrettyTable printStatistics(PrintStream out, PrintStream err) {
71         out.printf("Started at %s. Finished at %s. Run time: %s%n",
72                 Utils.formatDate(timeStarted()),
73                 Utils.formatDate(timeEnded()),
74                 Utils.prettyDate(runTime()));
75
76         return null;
77     }
78
79     /**
80      * Basic implementation of {@link ISolverStatistics} then just
81      * save time started and time ended
82      */
83     record Basic(long timeStarted, long timeEnded) implements ISolverStatistics {
84
85         @Override
86         public int totalStateExplored() {
87             return -1;
88         }
89
90         @Override
91         public long stateExploredPerSeconds() {
92             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;
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);
```

```
46
47          for (Level level : this.levels) {
48              level.pack = this;
49          }
50      }
51
52      public void writeSolutions(Path out) throws IOException, JsonException {
53          if (out == null) {
54              out = Path.of(sourcePath.toString() + ".solutions.json.gz");
55          }
56
57          boolean write = false;
58          for (Level level : levels) {
59              if (level.hasReport()) {
60                  write = true;
61              }
62          }
63
64          if (!write) {
65              return;
66          }
67
68          try (OutputStream os = Files.newOutputStream(out, StandardOpenOption.CREATE,
            ↪  StandardOpenOption.TRUNCATE_EXISTING)) {
69              BufferedWriter bw = new BufferedWriter(
70                      new OutputStreamWriter(new GZIPOutputStream(os)));
71
72              JsonPrettyWriter jpw = new JsonPrettyWriter(bw);
73
74              jpw.beginObject();
75              if (name != null) {
76                  jpw.field("pack", name);
77              } else {
78                  jpw.nullField("pack");
79              }
80              if (author != null) {
81                  jpw.field("author", author);
82              } else {
83                  jpw.nullField("author");
84              }
85
86              for (Level level : levels) {
87                  if (level.hasReport()) {
88
89                      jpw.key(String.valueOf(level.getIndex()));
90                      jpw.beginArray();
91
92                      level.writeSolutions(jpw);
93
94                      jpw.endArray();
95                  }
96              }
97
98              jpw.endObject();
99              jpw.close();
100         }
101     }
102
103     public void readSolutions(Path in) throws IOException, JsonException {
104         if (Files.notExists(in)) {
105             return;
106         }
107
```

```
108         try (InputStream is = Files.newInputStream(in)) {
109             BufferedReader br = new BufferedReader(
110                     new InputStreamReader(new GZIPInputStream(is)));
111 
112             JsonReader jr = new JsonReader(br);
113 
114             jr.beginObject();
115             jr.assertKeyEquals("pack");
116 
117             String pack;
118             if (jr.hasNextString()) {
119                 pack = jr.nextString();
120             } else {
121                 jr.nextNull();
122                 pack = null;
123             }
124 
125             jr.assertKeyEquals("author");
126             String author;
127             if (jr.hasNextString()) {
128                 author = jr.nextString();
129             } else {
130                 jr.nextNull();
131                 author = null;
132             }
133 
134             if (Objects.equals(pack, name) && Objects.equals(author, this.author)) {
135 
136                 while (!jr.isObjectEnd()) {
137                     int level = Integer.parseInt(jr.nextKey());
138 
139                     Level l = levels.get(level);
140                     jr.beginArray();
141 
142                     while (!jr.isArrayEnd()) {
143                         jr.beginObject();
144                         l.addSolverReport(SolverReport.fromJson(jr, l));
145                         jr.endObject();
146                     }
147 
148                     jr.endArray();
149                 }
150 
151                 jr.endObject();
152             }
153             jr.close();
154         }
155     }
156 
157     /**
158      * Returns the name of the pack
159      *
160      * @return the name of the pack
161      */
162     public String name() {
163         return name;
164     }
165 
166     /**
167      * Returns the author of the pack
168      *
169      * @return pack's author
170      */
```

```
171     public String author() {
172         return author;
173     }
174 
175     /**
176      * Returns all levels that are in this pack
177      *
178      * @return levels of this pack
179      */
180     public List<Level> levels() {
181         return levels;
182     }
183 
184     /**
185      * Returns the level at the specified index
186      *
187      * @param index the index of the level
188      * @return the level at the specified index
189      * @throws IndexOutOfBoundsException if the index is out of range
190      */
191     public Level getLevel(int index) {
192         return levels.get(index);
193     }
194 
195     /**
196      * Returns the number of level in this pack
197      *
198      * @return the number of level in this pack
199      */
200     public int nLevels() {
201         return levels.size();
202     }
203 
204     /**
205      * Returns the location of the file describing this pack. This is used for writing solutions
206      *
207      * @return the location of the file describing this pack
208      * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
209      */
210     public Path getSourcePath() {
211         return sourcePath;
212     }
213 
214     /**
215      * Sets the location of the file describing this pack. This is used for writing solutions
216      *
217      * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
218      */
219     public void setSourcePath(Path sourcePath) {
220         this.sourcePath = sourcePath;
221     }
222 }
```

Hotspots

```
1 package fr.valax.sokoshell.solver;
2 
3 import fr.valax.sokoshell.solver.board.Board;
4 import fr.valax.sokoshell.solver.board.Direction;
5 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6 
7 import java.util.ArrayDeque;
```

```java
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()) {
                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();
```

```
134         }
135         baseCrate.addCrate();
136 
137         return countTrue(accessible);
138     }
139 
140     private int countTrue(boolean[] array) {
141         int n = 0;
142 
143         for (int i = 0; i < array.length; i++) {
144             if (array[i]) {
145                 n++;
146             }
147         }
148 
149         return n;
150     }
151 
152     public boolean isHotspot(int crate, int blockingCrate) {
153         return hotspot[crate][blockingCrate];
154     }
155 
156     private record State(TileInfo player, TileInfo crate) {}
157 }
```