

fr.valax.sokoshell.solver	
└─ pathfinder	
└─ PlayerAStar	2
└─ AStarMarkSystem	3
└─ AbstractAStar	5
└─ CratePlayerAStar	8
└─ Node	9
└─ CrateAStar	10
└─ collections	
└─ Node	13
└─ SolverCollection	14
└─ MinHeap	14
└─ SolverPriorityQueue	17
└─ heuristic	
└─ AbstractHeuristic	18
└─ GreedyHeuristic	18
└─ SimpleHeuristic	21
└─ Heuristic	21
└─ board	
└─ tiles	
└─ MutableTileInfo	22
└─ GenericTileInfo	25
└─ ImmutableTileInfo	28
└─ TileInfo	29
└─ Tile	35
└─ mark	
└─ HeavyweightMarkSystem	36
└─ FixedSizeMarkSystem	36
└─ Mark	38
└─ DefaultMark	38
└─ MarkSystem	39
└─ AbstractMarkSystem	40
└─ Tunnel	40
└─ ImmutableBoard	44
└─ Move	46
└─ Room	47
└─ MutableBoard	48
└─ Direction	74
└─ GenericBoard	76
└─ Board	78
└─ State	85
└─ ReachableTiles	89
└─ Solver	89
└─ DeadlockTable	90
└─ AStarSolver	98
└─ Corral	100
└─ BruteforceSolver	107
└─ Tracker	110
└─ AbstractSolver	111
└─ SolverParameter	119
└─ Trackable	123
└─ CorralDetector	124
└─ WeightedState	132
└─ Level	133
└─ SolverReport	143
└─ FESS0Solver	154
└─ SolverParameters	157
└─ FreezeDeadlockDetector	159
└─ ISolverStatistics	161
└─ Pack	162
└─ Hotspots	166

1 fr.valax.sokoshell.solver

1.1 pathfinder

PlayerAStar

```
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
7 import java.util.PriorityQueue;
8
9 /**
10  * An 'A*' that can find a path between a start position and an end position for a player.
11  * It uses a local mark system.
12  */
13 public class PlayerAStar extends AbstractAStar {
14
15     private final int boardWidth;
16     private final AStarMarkSystem markSystem;
17     private final Node[] nodes;
18
19     public PlayerAStar(Board board) {
20         super(new PriorityQueue<>(board.getWidth() * board.getHeight()));
21         this.boardWidth = board.getWidth();
22         markSystem = new AStarMarkSystem(board.getWidth() * board.getHeight());
23         nodes = new Node[board.getHeight() * board.getWidth()];
24
25         for (int i = 0; i < nodes.length; i++) {
26             nodes[i] = new Node();
27         }
28     }
29
30     private int toIndex(TileInfo player) {
31         return player.getY() * boardWidth + player.getX();
32     }
33
34     @Override
35     protected void init() {
36         markSystem.unmarkAll();
37         queue.clear();
38     }
39
40     @Override
41     protected void clean() {
42
43     }
44
45     @Override
46     protected Node initialNode() {
47         int i = toIndex(playerStart);
48
49         Node init = nodes[i];
50         init.setInitial(playerStart, null, heuristic(playerStart));
51         return init;
52     }
53
54     @Override
55     protected Node processMove(Node parent, Direction dir) {
56         TileInfo player = parent.getPlayer();
57         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
67         ↪ the queue, therefore node.getExpectedDist() is valid
68         if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {
69             node.changeParent(parent);
70             decreasePriority(node);
71         }
72
73         return null;
74     } else {
75         markSystem.mark(i);
76         node.set(parent, dest, null, heuristic(dest));
77         return node;
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() {
81         mark = AStarMarkSystem.this.mark - 1;
82     }
83
84     public void setVisited() {
85         mark = AStarMarkSystem.this.mark;
86     }
87
88     @Override
89     public void unmark() {
90         mark = AStarMarkSystem.this.mark - 2;
91     }
92
93     @Override
94     public boolean isMarked() {
95         return mark == AStarMarkSystem.this.mark - 1;
96     }
97
98     public boolean isVisited() {
99         return mark == AStarMarkSystem.this.mark;
100     }
101
102     @Override
103     public MarkSystem getMarkSystem() {
104         return AStarMarkSystem.this;
105     }
106 }
107 }

```

AbstractAStar

```

1  package fr.valax.sokoshell.solver.pathfinder;
2
3  import fr.valax.sokoshell.solver.board.Direction;
4  import fr.valax.sokoshell.solver.board.Move;
5  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
7  import java.util.PriorityQueue;
8
9  /**
10   * Abstract implementation of A*.
11   */
12  public abstract class AbstractAStar {
13
14      protected TileInfo playerStart;
15      protected TileInfo crateStart;
16      protected TileInfo playerDest;
17      protected TileInfo crateDest;
18
19      protected final PriorityQueue<Node> queue;
20
21      public AbstractAStar(PriorityQueue<Node> queue) {
22          this.queue = queue;
23      }
24
25      /**
26       * @return true if path exists
27       * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
28       */
29      public boolean hasPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart,
30          ↪ TileInfo crateDest) {
31          return findPath(playerStart, playerDest, crateStart, crateDest) != null;
32      }
33  }

```

```

31 }
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,
39 ↪ TileInfo crateStart, TileInfo crateDest) {
40     Node end = findPath(playerStart, playerDest, crateStart, crateDest);
41
42     if (end == null) {
43         return null;
44     }
45
46     Node current = end;
47     while (current.getParent() != null) {
48         Node last = current.getParent();
49
50         TileInfo lastPlayer = last.getPlayer();
51         TileInfo currPlayer = current.getPlayer();
52         Direction dir = Direction.of(currPlayer.getX() - lastPlayer.getX(),
53 ↪ currPlayer.getY() - lastPlayer.getY());
54
55         boolean moved = crateStart != null &&
56 ↪ !current.getCrate().isAt(last.getCrate());
57         current.setMove(Move.of(dir, moved));
58
59         current = last;
60     }
61
62     return end;
63 }
64
65 /**
66  * Find a path between (playerStart, crateStart) and (playerDest, crateDest).
67  * The returned node may be cached by the implementation. Therefore, if you
68  * want to keep the path in memory, you need to copy the path.
69  *
70  * @param playerStart player start
71  * @param playerDest player dest
72  * @param crateStart crate start
73  * @param crateDest crate dest
74  * @return the shortest path as a linked list in reverse.
75  */
76 public Node findPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart,
77 ↪ TileInfo crateDest) {
78     this.playerStart = playerStart;
79     this.crateStart = crateStart;
80     this.playerDest = playerDest;
81     this.crateDest = crateDest;
82
83     init();
84     Node n = initialNode();
85     queue.offer(n);
86
87     // int c = 0;
88     Node end = null;
89     while (!queue.isEmpty()) {
90         Node node = queue.poll();
91
92         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
8   * ↪ dest):
9   * the player moves a crate from 'crate start' to 'crate dest' and then moves to 'player
10  * ↪ dest'.
11  */
12  public class CratePlayerAStar extends CrateAStar {
13
14      public CratePlayerAStar(Board board) {
15          super(board);
16      }
17
18      @Override
19      protected boolean isEndNode(Node node) {
20          return node.getPlayer().isAt(playerDest) && node.getCrate().isAt(crateDest);
21      }
22
23      @Override
24      protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
25          /**
26           * Try to first move the player near the crate
27           * Then push the crate to his destination
28           * Finally moves the player to his destination
29           */
30          int remaining = newCrate.manhattanDistance(crateDest);
31          if (remaining == 0) {
32              remaining = newPlayer.manhattanDistance(playerDest);
33          } else {
34              if (newPlayer.manhattanDistance(newCrate) > 1) {
35                  remaining += newPlayer.manhattanDistance(newCrate);
36              }
37
38              remaining += crateDest.manhattanDistance(playerDest);
39          }
40
41          return remaining;
42      }
43  }

```



```

40     }
41 }

```

Node

```

1 package fr.valax.sokoshell.solver.pathfinder;
2
3 import fr.valax.sokoshell.solver.board.Move;
4 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
5
6 import java.util.Objects;
7
8 /**
9  * A node in A*
10  */
11 public class Node implements Comparable<Node> {
12
13     private Node parent;
14     private int dist;
15     private int heuristic;
16     private TileInfo player;
17     private TileInfo crate;
18     private Move move;
19
20     private int expectedDist;
21
22     public Node() {
23     }
24
25     public Node(Node parent,
26                 int dist, int heuristic,
27                 TileInfo player, TileInfo crate, Move move) {
28         this.parent = parent;
29         this.dist = dist;
30         this.heuristic = heuristic;
31         this.player = player;
32         this.crate = crate;
33         this.move = move;
34     }
35
36     public void setInitial(TileInfo player, TileInfo crate, int heuristic) {
37         parent = null;
38         dist = 0;
39         this.heuristic = heuristic;
40         this.player = player;
41         this.crate = crate;
42
43         expectedDist = heuristic;
44     }
45
46     public void set(Node parent, TileInfo player, TileInfo crate, int heuristic) {
47         this.parent = parent;
48         this.dist = parent.dist + 1;
49         this.heuristic = heuristic;
50         this.player = player;
51         this.crate = crate;
52
53         expectedDist = dist + heuristic;
54     }
55
56     public void changeParent(Node newParent) {
57         this.parent = newParent;

```

```

58         this.dist = newParent.dist + 1;
59
60         expectedDist = dist + heuristic;
61     }
62
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
7 import java.util.PriorityQueue;
8
9 /**
10  * Moves a crate from a start position to a destination.
11  */
12 public class CrateAStar extends AbstractAStar {
13
14     private final int boardWidth;
15     private final int area;
16
17     private final AStarMarkSystem markSystem;
18     private final Node[] nodes;
19
20     public CrateAStar(Board board) {
21         super(new PriorityQueue<>((2 * board.getWidth() * board.getHeight())));
22         this.boardWidth = board.getWidth();
23
24         area = board.getWidth() * board.getHeight();
25         markSystem = new AStarMarkSystem(area * area);
26
27         nodes = new Node[area * area];
28
29         for (int i = 0; i < nodes.length; i++) {
30             nodes[i] = new Node();
31         }
32     }
33
34     private int toIndex(TileInfo player, TileInfo crate) {
35         return (player.getY() * boardWidth + player.getX()) * area + crate.getY() *
36             ↪ boardWidth + crate.getX();
37     }
38
39     @Override
40     protected void init() {
41         markSystem.unmarkAll();
42         queue.clear();
43         crateStart.removeCrate();
44     }
45
46     @Override
47     protected void clean() {
48         crateStart.addCrate();
49     }
50
51     @Override
52     protected Node initialNode() {
53         int i = toIndex(playerStart, crateStart);
54
55         Node init = nodes[i];
56         init.setInitial(playerStart, crateStart, heuristic(playerStart, crateStart));
57         return init;
58     }
59
60     @Override
61     protected Node processMove(Node parent, Direction dir) {

```

```

61     TileInfo player = parent.getPlayer();
62     TileInfo crate = parent.getCrate();
63     TileInfo playerDest = player.adjacent(dir);
64     TileInfo crateDest = crate;
65
66     if (playerDest.isAt(crate)) {
67         crateDest = playerDest.adjacent(dir);
68
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) {
132         h += newPlayer.manhattanDistance(newCrate);
133     }
134
135     return h;
136 }
137 }

```

1.2 collections

Node

```

1 package fr.valax.sokoshell.solver.collections;
2
3 public class Node<E> {
4
5     protected Node<E> next;
6     protected E value;
7
8     public Node(E value) {
9         this.value = value;
10    }
11
12    /**
13     * Detach this node from the linked list. After this call
14     * {@link #next()} will return null. If any node has for next
15     * this node, it won't be detached from these nodes.
16     *
17     * @return next node
18     */
19    public Node<E> detach() {
20        Node<E> oldNext = next;
21        next = null;
22        return oldNext;
23    }
24
25    /**
26     * Makes the specified node the previous node of this node.
27     *
28     * @param node new parent
29     */
30    public void attach(Node<E> node) {
31        node.next = this;
32    }
33
34    public Node<E> next() {
35        return next;
36    }
37
38    public E getValue() {
39        return value;
40    }
41
42    public void setValue(E value) {

```

```

43         this.value = value;
44     }
45 }

```

SolverCollection

```

1 package fr.valax.sokoshell.solver.collections;
2
3 import fr.valax.sokoshell.solver.State;
4
5 public interface SolverCollection<T extends State> {
6
7     void clear();
8
9     boolean isEmpty();
10
11     int size();
12
13     void addState(T state);
14
15     T popState();
16
17     T peekState();
18
19     T peekAndCacheState();
20
21     T cachedState();
22 }

```

MinHeap

```

1 package fr.valax.sokoshell.solver.collections;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.List;
6
7 public class MinHeap<T> {
8
9     /**
10      * Array of nodes.
11      */
12     protected final List<Node<T>> nodes;
13
14     protected int currentSize;
15
16     public MinHeap() {
17         nodes = new ArrayList<>();
18         currentSize = -1;
19     }
20
21     /**
22      * Creates a min heap of fixed capacity.
23      * This has 2 major consequences :
24      * <ul>
25      *     <li>this constructor instantiates empty object in each of the cases of the min
26      ↪ heap array</li>
27      *     <li>When {@link MinHeap#add(Object, int)} is called, no element is created nor
28      ↪ added : the case where the
29      *         new element goes is only updated with the new object values.</li>
30      * </ul>

```

```

29     * @param capacity The (fixed) capacity of the heap
30     */
31     public MinHeap(int capacity) {
32         nodes = new ArrayList<>(capacity);
33         for (int i = 0; i < capacity; i++) {
34             nodes.add(i, new Node<T>());
35         }
36         currentSize = 0;
37     }
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     }
101     moveNodeDown(0);
102     return content;
103 }
104
105 public T peek() {
106     return nodes.get(0).content();
107 }
108
109 public void clear() {
110     if (currentSize == -1) {
111         nodes.clear();
112     } else {
113         currentSize = 0;
114     }
115 }
116
117 public boolean isEmpty() {
118     return currentSize == -1 ? nodes.isEmpty() : (currentSize == 0);
119 }
120
121 public int size() {
122     return currentSize == -1 ? nodes.size() : currentSize;
123 }
124
125 /**
126  * Min heap (state, priority) couple.
127  */
128 protected static final class Node<T> {
129     private T content;
130     private int priority;
131
132     public Node() {
133         set(null, Integer.MAX_VALUE);
134     }
135
136     public Node(T content, int priority) {
137         set(content, priority);
138     }
139
140     public boolean hasPriorityOver(Node<T> o) {
141         return priority < o.priority;
142     }
143
144     @Override
145     public String toString() {
146         return String.format("Node[priority=%d]", priority);
147     }
148
149     public void set(T content, int priority) {
150         this.content = content;
151         this.priority = priority;
152     }

```



```

153
154     public T content() {
155         return content;
156     }
157
158     public void setContent(T content) {
159         this.content = content;
160     }
161
162     public int priority() {
163         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>
7  * ↪ order, i.e. the element returned
8  * by {@link SolverPriorityQueue#popState()} with the <strong>LOWEST</strong> priority.
9  */
10 public class SolverPriorityQueue implements SolverCollection<WeightedState> {
11
12     /**
13      * @implNote We use a min heap collection.
14      */
15     private final MinHeap<WeightedState> heap = new MinHeap<>();
16
17     private WeightedState cachedState;
18
19     @Override
20     public void addState(WeightedState state) {
21         heap.add(state, state.weight());
22     }
23
24     @Override
25     public WeightedState popState() {
26         return heap.pop();
27     }
28
29     @Override
30     public WeightedState peekState() {
31         return heap.peak();
32     }
33
34     @Override
35     public WeightedState peekAndCacheState() {
36         cachedState = popState();
37         return cachedState;
38     }
39
40     @Override
41     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
8  ↪ class each implementing
9  * different heuristic calculation methods.
10 */
11 public abstract class AbstractHeuristic implements Heuristic {
12
13     protected final Board board;
14
15     public AbstractHeuristic(Board board) {
16         this.board = board;
17     }
18 }

```

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
9  ↪ href="http://sokobano.de/wiki/index.php?title=Solver#Greedy_approach">this article</a>
10 */
11 public class GreedyHeuristic extends AbstractHeuristic {
12
13     private final LinkedList list;
14
15     public GreedyHeuristic(Board board) {
16         super(board);
17         final int n = board.getTargetCount();
18     }
19 }

```

```

17     list = new LinkedList(n);
18 }
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;
89     }
90     head = newHead;
91
92     size++;
93 }
94
95 public void remove(Node node) {
96     if (node == head) {
97         head = node.next;
98
99         if (head != null) {
100             head.previous = null;
101         }
102     } else {
103         node.previous.next = node.next;
104
105         if (node.next != null) {
106             node.next.previous = node.previous;
107         }
108     }
109
110     size--;
111 }
112
113 public Node getHead() {
114     return head;
115 }
116 }
117
118 private static class Node {
119
120     private final LinkedList list;
121     private TileInfo crate;
122
123     private Node previous;
124     private Node next;
125
126     /**
127      * Index in crate's target remoteness
128      */
129     private int index = 0;
130
131     public Node(LinkedList list) {
132         this.list = list;
133     }
134
135     public void set(TileInfo tile) {
136         crate = tile;
137         index = 0;
138     }
139
140     public void remove() {

```

```

141         list.remove(this);
142     }
143
144     public Node nextNode() {
145         return next;
146     }
147
148     public TileInfo getCrate() {
149         return crate;
150     }
151
152     public TileInfo.TargetRemoteness getNearestNotAttributedTarget() {
153         TileInfo.TargetRemoteness[] remoteness = crate.getTargets();
154
155         Board b = crate.getBoard();
156         while (b.getAt(remoteness[index].index()).isMarked()) {
157             index++;
158         }
159
160         return remoteness[index];
161     }
162 }
163

```

SimpleHeuristic

```

1 package fr.valax.sokoshell.solver.heuristic;
2
3 import fr.valax.sokoshell.solver.State;
4 import fr.valax.sokoshell.solver.board.Board;
5
6 /**
7  * According to <a
8  ↪ href="http://sokobano.de/wiki/index.php?title=Solver#Simple_Lower_Bound">this
9  ↪ article</a>
10  */
11 public class SimpleHeuristic extends AbstractHeuristic {
12
13     public SimpleHeuristic(Board board) {
14         super(board);
15     }
16
17     /**
18      * Sums the distances to the nearest goal of each of the crates of the state.
19      */
20     public int compute(State s) {
21         int h = 0;
22         for (int i : s.cratesIndices()) {
23             h += board.getAt(i).getNearestTarget().distance();
24         }
25         return h;
26     }
27 }

```

Heuristic

```

1 package fr.valax.sokoshell.solver.heuristic;
2
3 import fr.valax.sokoshell.solver.State;
4
5 /**

```

```

6  * Heuristic computing class for guided-search (e.g. A*)
7  */
8  public interface Heuristic {
9
10     /**
11      * Computes the heuristic of the given state.
12      * @param s the state to compute the heuristic
13      * @return the heuristic of the state
14      */
15     int compute(State s);
16
17 }

```

1.4 board

1.4.1 tiles

MutableTileInfo

```

1  package fr.valax.sokoshell.solver.board.tiles;
2
3  import fr.valax.sokoshell.solver.State;
4  import fr.valax.sokoshell.solver.board.MutableBoard;
5  import fr.valax.sokoshell.solver.board.Room;
6  import fr.valax.sokoshell.solver.board.Tunnel;
7  import fr.valax.sokoshell.solver.board.mark.Mark;
8
9  /**
10   * Mutable implementation of {@link TileInfo}.
11   *
12   * This class extends {@link GenericTileInfo} and implements the setters methods defined
13   ↪ in
14   * {@link TileInfo}.
15   * It also implements getters and setters for the 'solver-intended' properties.
16   *
17   * @see TileInfo
18   * @see GenericTileInfo
19   */
20  public class MutableTileInfo extends GenericTileInfo {
21
22      private final MutableBoard board;
23
24      // Static information
25      protected boolean deadTile;
26
27      /**
28       * The tunnel in which this tile is. A Tile is either in a room or in a tunnel
29       */
30      protected Tunnel tunnel;
31      // contains for each direction, where is the outside of the tunnel from this tile
32      protected Tunnel.Exit tunnelExit;
33      protected Room room;
34
35      /**
36       * Remoteness data from this tile to every target on the board.
37       */
38      protected TargetRemoteness[] targets;
39
40      /**
41       * Nearest target on the board.
42       */
43      protected TargetRemoteness nearestTarget;
44
45 }

```

```

44  /**
45   * The index of this crate in the {@link State#cratesIndices()} array
46   */
47  protected int crateIndex;
48
49
50  // Dynamic information
51  protected Mark reachable;
52  protected Mark mark;
53
54  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}
9  * ↪ implementations.
10  * It defines all the basic properties and their corresponding getters
11  * (position, tile, board, etc.)
12  *
13  * @see TileInfo
14  */
15 public abstract class GenericTileInfo implements TileInfo {
16
17     protected final Board board;
18
19     protected final int x;
20     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}.
9  *
10  * This class basically extends {@link GenericTileInfo}. It implements the setters methods
11  * ↪ defined in
12  * ↪ {@link TileInfo} by throwing an {@link UnsupportedOperationException}.
13  * ↪ It also implements the 'solver-intended' properties by always returning the default
14  * ↪ value: for instance, a
15  * ↪ {@link ImmutableTileInfo} is never a 'dead tile', so the {@link #isDeadTile} method
16  * ↪ will always return {@code false}.
17  * ↪ The same policy is applied for each property.
18  *
19  * @see TileInfo
20  * @see GenericTileInfo
21  */
22 public class ImmutableTileInfo extends GenericTileInfo {
23
24     public ImmutableTileInfo(ImmutableBoard board, Tile tile, int x, int y) {
25         super(board, tile, x, y);
26     }
27
28     public ImmutableTileInfo(TileInfo tileInfo) {
29         super(tileInfo);
30     }
31
32     // GETTERS //
33
34     @Override
35     public boolean isDeadTile() {
36         return false;
37     }
38
39     @Override
40     public boolean isReachable() {
41         return true;
42     }
43
44     @Override
45     public Tunnel getTunnel() {
46         return null;
47     }
48
49     @Override
50     public Tunnel.Exit getTunnelExit() {
51         return null;
52     }
53
54     @Override
55     public boolean isInATunnel() {
56         return false;
57     }
58 }

```

```

54     }
55
56     @Override
57     public Room getRoom() {
58         return null;
59     }
60
61     @Override
62     public boolean isInARoom() {
63         return false;
64     }
65
66     @Override
67     public boolean isMarked() {
68         return false;
69     }
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
12  * ↪ need to manage tiles,
13  * for instance:
14  * <ul>
15  *     <li>the position</li>
16  *     <li>the {@link Tile}</li>
17  * </ul>
18  * It defines a set of high-level interactions functions.
19  *
20  * @see Board
21  */
22 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    /**
101    * Returns {@code true} if this tile is at the position (x; y)
102    * @param x x location
103    * @param y y location
104    * @return {@code true} if this tile is at the position (x; y)}
105    */
106    default boolean isAt(int x, int y) {
107        return x == getX() && y == getY();
108    }
109
110    /**
111    * Returns the direction between this tile and other.
112    *
113    * @param other 'other' tile
114    * @return the direction between this tile and other
115    */
116    default Direction direction(TileInfo other) {
117        return Direction.of(other.getX() - getX(), other.getY() - getY());
118    }
119
120    /**
121    * Returns the distance of manhattan between this tile and other
122    *
123    * @param other 'other' tile
124    * @return the distance of manhattan between this tile and other
125    */
126    default int manhattanDistance(TileInfo other) {
127        return Math.abs(getX() - other.getX()) + Math.abs(getY() - other.getY());
128    }
129
130    /**
131    * @return {@code true} if this tile is a dead tile
132    * @see MutableBoard#computeDeadTiles()
133    */
134    boolean isDeadTile();
135
136    /**
137    * @return {@code true} if this tile is reachable by the player.
138    * @see MutableBoard#findReachableCases(int)
139    */
140    boolean isReachable();
141
142    /**
143    * Returns the tunnel in which this tile is
144    *
145    * @return the tunnel in which this tile is

```

```

146     */
147     Tunnel getTunnel();
148
149     /**
150     * Returns the {@link Tunnel.Exit} object associated with this tile info.
151     * If the tile isn't in a tunnel, it returns null
152     *
153     * @return the {@link Tunnel.Exit} object associated with this tile info or {@code
↪ null
154     * @see Tunnel.Exit
155     */
156     Tunnel.Exit getTunnelExit();
157
158     /**
159     * Returns {@code true} if this tile info is in a tunnel
160     *
161     * @return {@code true} if this tile info is in a tunnel
162     */
163     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     // SETTERS //
246
247     /**
248     * If this was a floor, this is now a crate
249     * If this was a target, this is now a crate on target
250     * @throws UnsupportedOperationException if the {@code addCrate} operation isn't
251     * supported by this TileInfo
252     */
253     void addCrate();
254
255     /**
256     * If this was a crate, this is now a floor
257     * If this was a crate on target, this is now a target
258     * @throws UnsupportedOperationException if the {@code removeCrate} operation isn't
259     * supported by this TileInfo
260     */
261     void removeCrate();
262
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     /**
22      * Mark or not the object. After this method is called, {@link #isMarked()}
23      * will return {@code marked}
24      */
25     default void setMarked(boolean marked) {
26         if (marked) {
27             mark();
28         } else {
29             unmark();
30         }
31     }
32
33     /**
34      * @return true is the object is marked
35      */
36     boolean isMarked();
37
38     /**
39      * @return the {@link MarkSystem} associated with this mark
40      */
41     MarkSystem getMarkSystem();
42 }

```

DefaultMark

```

1 package fr.valax.sokoshell.solver.board.mark;
2
3 public class DefaultMark implements Mark {
4
5     private final MarkSystem markSystem;
6     private int mark;
7
8     public DefaultMark(MarkSystem markSystem) {
9         this.markSystem = markSystem;
10        unmark();

```

```

11     }
12
13     @Override
14     public void mark() {
15         mark = markSystem.getMark();
16     }
17
18     @Override
19     public void unmark() {
20         mark = markSystem.getMark() - 1;
21     }
22
23     @Override
24     public boolean isMarked() {
25         return mark == markSystem.getMark();
26     }
27
28     @Override
29     public MarkSystem getMarkSystem() {
30         return markSystem;
31     }
32 }

```

MarkSystem

```

1 package fr.valax.sokoshell.solver.board.mark;
2
3 /**
4  * <p>
5  *     A MarkSystem is used by dfs/bfs/others algorithm to avoid checking twice an object.
6  *     With a MarkSystem, you don't need to unmark all visited objects
7  *     {@link Mark} associated with this system can be created using {@link #newMark()}.
8  * </p>
9  * <h2>How it works</h2>
10 * <p>
11 *     A mark have a value, the same for a MarkSystem. A mark is marked if it value is
12 *     ↪ equals
13 *     to the value of the MarkSystem. So, to unmark all mark, you just have to increase
14 *     the MarkSystem's value.
15 * </p>
16 * @see Mark
17 * @author PoulpoGaz
18 */
19 public interface MarkSystem {
20     /**
21      * Create a new mark associated with this MarkSystem.
22      * The mark is by default unmarked
23      * @return a new mark
24      */
25     Mark newMark();
26
27     /**
28      * Unmark all marks
29      */
30     void unmarkAll();
31
32     /**
33      * Set the 'selected' mark to 0 and unmark all Mark
34      */
35     void reset();
36 }

```

```

37     /**
38      * @return the selected mark.
39      */
40     int getMark();
41 }

```

AbstractMarkSystem

```

1 package fr.valax.sokoshell.solver.board.mark;
2
3 /**
4  * Contains the basic for all mark system
5  */
6 public abstract class AbstractMarkSystem implements MarkSystem {
7
8     /**
9      * A mark is marked if it's value is equals to this field
10    */
11    protected int mark;
12
13    @Override
14    public Mark newMark() {
15        return new DefaultMark(this);
16    }
17
18    @Override
19    public void unmarkAll() {
20        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
160                 ↪ "closed - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
161                     .formatted(start.getX(), start.getY(),
162                               end.getX(), end.getY(),
163                               endOut.getX(), endOut.getY(),
164                               playerOnlyTunnel, isOneway);
165         } else if (endOut == null) {
166             return
167                 ↪ " (%d; %d) - (%d; %d) --> (%d; %d) - closed. only player? %s. one way? %s"
168                     .formatted(startOut.getX(), startOut.getY(),
169                               start.getX(), start.getY(),
170                               end.getX(), end.getY(),
171                               playerOnlyTunnel, isOneway);
172         } else {
173             return
174                 ↪ " (%d; %d) - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
175                     .formatted(startOut.getX(), startOut.getY(),
176                               start.getX(), start.getY(),
177                               end.getX(), end.getY(),
178                               endOut.getX(), endOut.getY(),
179                               playerOnlyTunnel, isOneway);
180         }
181     }
182
183     /**
184     * Added to every tile that is inside a tunnel.
185     * It contains for each direction where is the exit:
186     * if you push a crate inside the tunnel to the left, the
187     * method {@link #getExit(Direction)} will return where you will
188     * be after pushing the crate until you aren't outside the tunnel.
189     *
190     * @implNote This object isn't immutable but is assumed as
191     * immutable by
192     ↪ MutableBoard.StaticBoard#linkTunnelsRoomsAndTileInfos(MutableBoard.StaticTile[][])
193     */
194     public static class Exit {
195
196         private TileInfo leftExit;
197         private TileInfo upExit;
198         private TileInfo rightExit;
199         private TileInfo downExit;
200
201         public Exit() {
202

```

```

199
200     public Exit(TileInfo leftExit, TileInfo upExit, TileInfo rightExit, TileInfo
    ↪     downExit) {
201         this.leftExit = leftExit;
202         this.upExit = upExit;
203         this.rightExit = rightExit;
204         this.downExit = downExit;
205     }
206
207     public TileInfo getExit(Direction dir) {
208         return switch (dir) {
209             case LEFT -> leftExit;
210             case UP -> upExit;
211             case RIGHT -> rightExit;
212             case DOWN -> downExit;
213         };
214     }
215
216     public TileInfo getLeftExit() {
217         return leftExit;
218     }
219
220     private void setLeftExit(TileInfo leftExit) {
221         this.leftExit = leftExit;
222     }
223
224     public TileInfo getUpExit() {
225         return upExit;
226     }
227
228     private void setUpExit(TileInfo upExit) {
229         this.upExit = upExit;
230     }
231
232     public TileInfo getRightExit() {
233         return rightExit;
234     }
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}
14  * to store the board content
15  * in {@link GenericBoard#content}. As it is immutable, it implements the setters methods
16  * always throws a
17  * {@link UnsupportedOperationException} when such a method is called.
18  *
19  * @see Board
20  * @see GenericBoard
21  * @see TileInfo
22  */
23 public class ImmutableBoard extends GenericBoard {
24
25     public ImmutableBoard(Tile[][] content, int width, int height) {
26         super(width, height);
27
28         this.content = new ImmutableTileInfo[height][width];
29
30         for (int y = 0; y < height; y++) {
31             for (int x = 0; x < width; x++) {
32                 this.content[y][x] = new ImmutableTileInfo(this, content[y][x], x, y);
33             }
34         }
35
36     public ImmutableBoard(Board other) {
37         super(other.getWidth(), other.getHeight());
38
39         this.content = new ImmutableTileInfo[height][width];
40
41         for (int y = 0; y < height; y++) {
42             for (int x = 0; x < width; x++) {
43                 this.content[y][x] = new ImmutableTileInfo(other.getAt(x, y));
44             }
45         }
46
47     }
48
49     // GETTERS //
50
51     @Override
52     public int getTargetCount() {
53         return 0;
54     }
55
56     @Override
57     public List<Tunnel> getTunnels() {
58         return null;
59     }
60
61     @Override
62     public List<Room> getRooms() {
63         return null;
64     }
65
66     @Override
67     public boolean isGoalRoomLevel() {
68         return false;
69     }

```

```

69     @Override
70     public MarkSystem getMarkSystem() {
71         return null;
72     }
73
74     @Override
75     public MarkSystem getReachableMarkSystem() {
76         return null;
77     }
78 }

```

Move

```

1  package fr.valax.sokoshell.solver.board;
2
3  /**
4   * An enumeration representing a move or a push in a solution. The {@code moveCrate} flag
5   * ↪ is needed to go back
6   * in {@link fr.valax.sokoshell.commands.level.SolutionCommand}
7   *
8   * DO NOT MODIFY ORDER OF VALUES WITHOUT REMAKING ALL SAVES
9   */
10 public enum Move {
11     LEFT("l", Direction.LEFT, false),
12     UP("u", Direction.UP, false),
13     DOWN("d", Direction.DOWN, false),
14     RIGHT("r", Direction.RIGHT, false),
15
16     LEFT_PUSH("L", Direction.LEFT, true),
17     UP_PUSH("U", Direction.UP, true),
18     RIGHT_PUSH("R", Direction.RIGHT, true),
19     DOWN_PUSH("D", Direction.DOWN, true);
20
21     private final String shortName;
22     private final Direction direction;
23     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
19      ↪ Board#isGoalRoomLevel()}
20      */
21     protected List<TileInfo> packingOrder;
22
23     // dynamic
24     // the index in packingOrder of the position of the next crate that will be pushed
25     ↪ inside the room
26     // negative if it is not possible because a crate isn't at the correct position
27     // or if the room isn't a goal room
28     protected int packingOrderIndex;
29
30     public Room() {
31
32     }
33
34     public void addTile(TileInfo tile) {
35         tiles.add(tile);
36
37         if (tile.isTarget()) {
38             targets.add(tile);
39         }
40     }
41
42     public List<TileInfo> getTiles() {
43         return tiles;
44     }
45
46     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
27  * ↪ internally uses {@link MutableTileInfo} to store the board content
28  * in {@link GenericBoard#content}.
29  *
30  * @see Board
31  * @see GenericBoard
32  * @see MutableTileInfo
33  */
34 @SuppressWarnings("ForLoopReplaceableByForEach")
35 public class MutableBoard extends GenericBoard {
36
37     private final MarkSystem markSystem = newMarkSystem(TileInfo::unmark);
38     private final MarkSystem reachableMarkSystem = newMarkSystem((t) ->
39         ↪ t.setReachable(false));
40
41     private int targetCount;
42
43     /**
44      * Tiles that can be 'target' or 'floor'
45      */
46     private TileInfo[] floors;
47
48     private final List<Tunnel> tunnels = new ArrayList<>();
49     private final List<Room> rooms = new ArrayList<>();
50
51     /**
52      * True if all rooms are goal room with only one entrance
53      */
54     private boolean isGoalRoomLevel;
55
56     private PlayerAStar playerAStar;
57     private CrateAStar crateAStar;
58     private CratePlayerAStar cratePlayerAStar;
59
60     private final CorralDetector corralDetector;
61
62     private StaticBoard staticBoard;
63
64     /**
65      * Creates a SolverBoard with the specified width, height and tiles
66      *
67      * @param content a rectangular matrix of size width * height. The first index is for
68      * ↪ the rows
69      * and the second for the columns
70      * @param width board width
71      * @param height board height
72      */
73     public MutableBoard(Tile[][] content, int width, int height) {
74         super(width, height);
75
76         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(),
149                     ↪ tile.getTargets().length));
150             }
151             tile.setNearestTarget(otherTile.getNearestTarget());
152
153             tile.setTunnel(tunnelMap.get(otherTile.getTunnel()));
154             tile.setRoom(roomMap.get(otherTile.getRoom()));
155             if (otherTile.getTunnelExit() != null) {
156                 tile.setTunnelExit(otherTile.getTunnelExit()); // it is immutable !
157             }
158         }
159     }
160
161     // link rooms and tunnels
162     for (Tunnel tunnel : other.getTunnels()) {
163         Tunnel newTunnel = tunnelMap.get(tunnel);
164         for (Room room : other.getRooms()) {
165             Room newRoom = roomMap.get(room);
166             newTunnel.addRoom(newRoom);
167             newRoom.addTunnel(newTunnel);
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) {
229         ↪ content[getY(index)][getX(index)].setTile(tile); }
230
231     /**
232      * Set at tile at (x, y)
233      *
234      * @param x x position in the board
235      * @param y y position in the board
236      * @throws IndexOutOfBoundsException if the position is outside the board
237      */
238     public void setAt(int x, int y, Tile tile) {
239         content[y][x].setTile(tile);
240     }
241
242     /**
243      * Puts the crates of the given state in the content array.
244      *
245      * @param state The state with the crates
246      */
247     public void addStateCrates(State state) {
248         int[] cratesIndices = state.cratesIndices();
249         for (int j = 0; j < cratesIndices.length; j++) {
250             int i = cratesIndices[j];
251             TileInfo crate = getAt(i);
252             crate.setCrateIndex(j);
253             crate.addCrate();
254         }
255     }
256
257     /**
258      * Removes the crates of the given state from the content array.
259      *

```

```

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     *
370     * @param consumer the consumer to apply
371     */
372     public void forEachNotWall(Consumer<TileInfo> consumer) {
373         for (TileInfo floor : floors) {
374             consumer.accept(floor);
375         }
376     }
377
378     public void computeTunnelStatus(State state) {
379         for (int i = 0; i < tunnels.size(); i++) {
380             tunnels.get(i).setCrateInside(false);

```

```

381     }
382
383     for (int i : state.cratesIndices()) {
384         Tunnel t = getAt(i).getTunnel();
385         if (t != null) {
386             // TODO: do the check but need to check if player is between two crates in
387             // ↪ a tunnel: see boxxle 53
388             /*if (t.crateInside()) { // THIS IS VERY IMPORTANT -> see tunnels
389                 throw new IllegalStateException();
390             }*/
391
392             t.setCrateInside(true);
393         }
394     }
395
396     public void computePackingOrderProgress(State state) {
397         if (!isGoalRoomLevel) {
398             return;
399         }
400
401         for (int i = 0; i < rooms.size(); i++) {
402             rooms.get(i).setPackingOrderIndex(0);
403         }
404
405         for (int i : state.cratesIndices()) {
406             TileInfo tile = getAt(i);
407
408             Room r = tile.getRoom();
409             if (r != null) {
410                 if (r.isGoalRoom() && tile.isCrate()) { // crate whereas a goal room must
411                     // ↪ contain crate on target
412                     r.setPackingOrderIndex(-1);
413                 }
414             }
415
416             for (int i = 0; i < rooms.size(); i++) {
417                 Room r = rooms.get(i);
418
419                 if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
420                     List<TileInfo> order = r.getPackingOrder();
421
422                     // find the first non crate on target tile
423                     // if the room is completed, then index is equals to -1
424                     int index = -1;
425                     for (int j = 0; j < order.size(); j++) {
426                         TileInfo tile = order.get(j);
427
428                         if (!tile.isCrateOnTarget()) {
429                             index = j;
430                             break;
431                         }
432                     }
433
434                     // checks that remaining aren't crate on target
435                     for (int j = index + 1; j < order.size(); j++) {
436                         TileInfo tile = order.get(j);
437
438                         if (tile.isCrateOnTarget()) {
439                             index = -1;
440                             break;

```

```

441         }
442     }
443
444     r.setPackingOrderIndex(index);
445 } else {
446     r.setPackingOrderIndex(-1);
447 }
448 }
449 }
450
451 // *****
452 // * ANALYSIS *
453 // *****
454
455 // * STATIC *
456
457 /**
458  * Detects the dead positions of a level. Dead positions are cases that make the level
↪ unsolvable
459  * when a crate is put on them.
460  * After this function has been called, to check if a given crate at (x,y) is a dead
↪ position,
461  * you can use {@link TileInfo#isDeadTile()} to check in constant time.
462  * The board <strong>MUST</strong> have <strong>NO CRATES</strong> for this function
↪ to work.
463  */
464 public void computeDeadTiles() {
465     // reset
466     forEachNotWall(tile -> tile.setDeadTile(true));
467
468     // loop
469     forEachNotWall((tile) -> {
470         if (!tile.isDeadTile()) {
471             return;
472         }
473
474         if (tile.anyCrate()) {
475             tile.setDeadTile(true);
476             return;
477         }
478
479         if (!tile.isTarget()) {
480             return;
481         }
482
483         findNonDeadCases(tile, null);
484     });
485 }
486 /**
487  * Discovers all the reachable cases from (x, y) to find dead positions.
488  */
489 private void findNonDeadCases(TileInfo tile, Direction lastDir) {
490     tile.setDeadTile(false);
491     for (Direction d : Direction.VALUES) {
492         if (d == lastDir) { // do not go backwards
493             continue;
494         }
495
496         final int nextX = tile.getX() + d.dirX();
497         final int nextY = tile.getY() + d.dirY();
498         final int nextNextX = nextX + d.dirX();
499         final int nextNextY = nextY + d.dirY();

```



```

500         if (getAt(nextX, nextY).isDeadTile() // avoids to check already processed
501             ↪ cases
502             && isTileEmpty(nextX, nextY)
503             && isTileEmpty(nextNextX, nextNextY)) {
504             findNonDeadCases(getAt(nextX, nextY), d.negate());
505         }
506     }
507 }
508
509 /**
510  * Find tunnels. A tunnel is something like this:
511  * <pre>
512  *     $$$$$$
513  *           $$$$$$
514  *     $$$$
515  *           $$$$$$$$
516  * </pre>
517  *
518  * A tunnel doesn't contain a target
519  */
520 public void findTunnels() {
521     tunnels.clear();
522
523     markSystem.unmarkAll();
524     forEachNotWall((t) -> {
525         if (t.isInATunnel() || t.isMarked() || t.isTarget()) {
526             return;
527         }
528
529         Tunnel tunnel = buildTunnel(t);
530
531         if (tunnel != null) {
532             tunnels.add(tunnel);
533         }
534     });
535 }
536
537 /**
538  * Try to create a tunnel that contains the specified tile.
539  *
540  * @param init a tile in the tunnel
541  * @return a tunnel that contains the tile or {@code null}
542  */
543 private Tunnel buildTunnel(TileInfo init) {
544     Direction pushDir1 = null;
545     Direction pushDir2 = null;
546
547     for (Direction dir : Direction.VALUES) {
548         TileInfo adj = init.adjacent(dir);
549
550         if (!adj.isSolid()) {
551             if (pushDir1 == null) {
552                 pushDir1 = dir;
553             } else if (pushDir2 == null) {
554                 pushDir2 = dir;
555             } else {
556                 return null; // too many direction
557             }
558         }
559     }
560 }

```

```

561     if (pushDir1 == null) { // all adjacents tiles are wall, ie init is alone, nerver
562         ↪ happen see LevelBuilder
563         return null;
564     } else if (pushDir2 == null) {
565         /*
566             We are in this case:
567             |$|
568             $| |$
569             */
570         Tunnel tunnel = new Tunnel();
571         tunnel.setStart(init);
572         tunnel.setEnd(init);
573         tunnel.setEndOut(init.adjacent(pushDir1));
574         init.setTunnel(tunnel);
575
576         growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
577         return tunnel;
578     } else {
579         /*
580             Either:
581             #| |#
582             Either:
583             |#|
584             #| |
585             */
586         boolean onlyPlayer = false;
587
588         if (pushDir1.negate() != pushDir2) {
589             /*
590                 First case:
591                 |#|
592                 #|i|
593                 | |#
594                 if init is like this, then this is a tunnel and a crate
595                 mustn't be pushed inside.
596
597                 Second case:
598                 |#|
599                 #|i|
600                 | |
601                 ie not tunnel
602             */
603             if (init.adjacent(pushDir1).adjacent(pushDir2).isSolid()) {
604                 onlyPlayer = true;
605             } else {
606                 return null;
607             }
608         }
609
610         Tunnel tunnel = new Tunnel();
611         tunnel.setEnd(init);
612         tunnel.setEndOut(init.adjacent(pushDir1));
613         tunnel.setPlayerOnlyTunnel(onlyPlayer);
614         init.setTunnel(tunnel);
615
616         growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
617         tunnel.setStart(tunnel.getEnd());
618         tunnel.setStartOut(tunnel.getEndOut());
619         growTunnel(tunnel, init.adjacent(pushDir2), pushDir2);
620
621         return tunnel;

```

```

622     }
623 }
624
625 /**
626  * Try to grow a tunnel by the end ie Tunnel#end and Tunnel#endOut are modified.
627  * The tile adjacent to pos according to -dir is assumed to
628  * be a part of a tunnel. So we are in the following situations:
629  * <pre>
630  *           $$$      $$$
631  *      $ $      $      $
632  *      @$      @$      @$
633  * </pre>
634  *
635  * @param pos position of the player
636  * @param dir the move the player did to go to pos
637  */
638 private void growTunnel(Tunnel t, TileInfo pos, Direction dir) {
639     pos.mark();
640
641     Direction leftDir = dir.left();
642     Direction rightDir = dir.right();
643     TileInfo left = pos.adjacent(leftDir);
644     TileInfo right = pos.adjacent(rightDir);
645     TileInfo front = pos.adjacent(dir);
646
647     if (!pos.isTarget()) {
648         pos.setTunnel(t);
649         if (left.isSolid() && right.isSolid() && front.isSolid()) {
650             t.setPlayerOnlyTunnel(true);
651             t.setEnd(pos);
652             t.setEndOut(null);
653             return;
654         } else if (left.isSolid() && right.isSolid()) {
655             if (front.isMarked()) {
656                 t.setEnd(pos);
657                 t.setEndOut(front);
658             } else {
659                 growTunnel(t, front, dir);
660             }
661             return;
662         } else if (right.isSolid() && front.isSolid()) {
663             t.setPlayerOnlyTunnel(true);
664             if (left.isMarked()) {
665                 t.setEnd(pos);
666                 t.setEndOut(left);
667             } else {
668                 growTunnel(t, left, leftDir);
669             }
670             return;
671         } else if (left.isSolid() && front.isSolid()) {
672             t.setPlayerOnlyTunnel(true);
673             if (right.isMarked()) {
674                 t.setEnd(pos);
675                 t.setEndOut(right);
676             } else {
677                 growTunnel(t, right, rightDir);
678             }
679             return;
680         }
681     }
682
683     pos.setTunnel(null);

```

```

684     pos.unmark();
685     t.setEndOut(pos);
686     t.setEnd(pos.adjacent(dir.negate()));
687 }
688
689 /**
690  * Finds room based on tunnel. Basically all tile that aren't in a tunnel are in room.
691  * This means that you need to call {@link #findTunnels()} before!
692  * A room that contains a target is a packing room.
693  */
694 public void findRooms() {
695     forEachNotWall((t) -> {
696         if (t.isInATunnel() || t.isInARoom()) {
697             return;
698         }
699
700         Room room = new Room();
701         expandRoom(room, t);
702         rooms.add(room);
703     });
704 }
705
706 private void expandRoom(Room room, TileInfo tile) {
707     room.addTile(tile);
708     tile.setRoom(room);
709
710     if (tile.isTarget()) {
711         room.setGoalRoom(true);
712     }
713
714     for (Direction dir : Direction.VALUES) {
715         TileInfo adj = tile.adjacent(dir);
716
717         if (!adj.isSolid()) {
718             if (!adj.isInATunnel() && !adj.isInARoom()) {
719                 expandRoom(room, adj);
720             } else if (adj.isInATunnel()) {
721                 // avoid add two times a tunnel to a room
722                 // It occurs when a tunnel has his two entrance
723                 // connected to a room
724                 if (room.tunnels == null || !room.tunnels.contains(adj.getTunnel())) {
725                     room.addTunnel(adj.getTunnel());
726                     adj.getTunnel().addRoom(room);
727                 }
728             }
729         }
730     }
731 }
732
733 /**
734  * Due to this, SokHard 49 can't be solved...
735  */
736 private void removeUselessTunnels() {
737     for (int i = 0; i < tunnels.size(); i++) {
738         Tunnel t = tunnels.get(i);
739         if (t.getStartOut() == null || t.getEndOut() == null) {
740             Room room = t.getRooms().get(0); // tunnel is linked to exactly one room
741             room.tunnels.remove(t); // detach the tunnel
742
743             if (room.tunnels.size() == 2 && room.tiles.size() == 1 &&
744                 ↪ !room.isGoalRoom()) {
745                 // room is now useless

```

```

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

```

```

807         t1.setEnd(t2.getEnd());
808         t1.setEndOut(t2.getEndOut());
809     } else {
810         t1.setEnd(t2.getStart());
811         t1.setEndOut(t2.getStartOut());
812     }
813 }
814
815 forEachNotWall((t) -> {
816     if (t.getTunnel() == t2) {
817         t.setTunnel(t1);
818     }
819 });
820
821 toAdd.setRoom(null);
822 toAdd.setTunnel(t1);
823 t1.setPlayerOnlyTunnel(t1.isPlayerOnlyTunnel() || t2.isPlayerOnlyTunnel());
824 t1.rooms.remove(room);
825 t2.rooms.remove(room);
826
827 for (Room r : t2.rooms) {
828     r.tunnels.remove(t2);
829     r.tunnels.add(t1);
830 }
831
832 t1.rooms.addAll(t2.rooms);
833 }
834
835 private void finishComputingTunnels() {
836     for (int i = 0; i < tunnels.size(); i++) {
837         Tunnel tunnel = tunnels.get(i);
838
839         // compute tunnel exits
840         tunnel.createTunnelExits();
841
842         // compute oneway property
843         if (tunnel.getStartOut() == null || tunnel.getEndOut() == null) {
844             tunnel.setOneway(true);
845         } else {
846             tunnel.getStart().addCrate();
847             corralDetector.findCorral(this, tunnel.getStartOut().getX(),
848                                     ↪ tunnel.getStartOut().getY());
849             tunnel.getStart().removeCrate();
850
851             tunnel.setOneway(!tunnel.getEndOut().isReachable());
852         }
853     }
854 }
855
856 /**
857  * Compute packing order. No crate should be on the board
858  */
859 public void tryComputePackingOrder() {
860     isGoalRoomLevel = rooms.size() > 1;
861
862     if (!isGoalRoomLevel) {
863         return;
864     }
865
866     for (int i = 0; i < rooms.size(); i++) {
867         Room r = rooms.get(i);
868         if (r.isGoalRoom() && r.getTunnels().size() != 1) {

```

```

868         isGoalRoomLevel = false;
869         break;
870     }
871 }
872
873 if (isGoalRoomLevel) {
874     for (Room r : rooms) {
875         if (r.isGoalRoom() && !computePackingOrder(r)) {
876             isGoalRoomLevel = false; // failed to compute packing order for a
877                                     ↪ room...
878             break;
879         }
880     }
881 }
882
883 /**
884  * The room must have only one entrance and a packing room
885  * @param room a room
886  */
887 private boolean computePackingOrder(Room room) {
888     markSystem.unmarkAll();
889
890     Tunnel tunnel = room.getTunnels().get(0);
891     TileInfo entrance;
892     TileInfo inRoom;
893     if (tunnel.getStartOut() != null && tunnel.getStartOut().getRoom() == room) {
894         entrance = tunnel.getStart();
895         inRoom = tunnel.getStartOut();
896     } else {
897         entrance = tunnel.getEnd();
898         inRoom = tunnel.getEndOut();
899     }
900
901     List<TileInfo> targets = room.getTargets();
902     for (TileInfo t : targets) {
903         t.addCrate();
904     }
905
906     List<TileInfo> packingOrder = new ArrayList<>();
907
908     List<TileInfo> frontier = new ArrayList<>();
909     List<TileInfo> newFrontier = new ArrayList<>();
910     frontier.add(entrance);
911
912     List<TileInfo> accessibleCrates = new ArrayList<>();
913     findAccessibleCrates(frontier, newFrontier, accessibleCrates);
914
915     while (!accessibleCrates.isEmpty()) {
916         boolean hasChanged = false;
917
918         for (int i = 0; i < accessibleCrates.size(); i++) {
919             TileInfo crate = accessibleCrates.get(i);
920             crate.removeCrate();
921             inRoom.addCrate();
922
923             if (crateAStar.hasPath(entrance, null, inRoom, crate)) {
924                 accessibleCrates.remove(i);
925                 i--;
926                 crate.unmark();
927                 crate.removeCrate();
928             }

```

```

929
930         // discover new accessible crates
931         frontier.add(crate);
932         findAccessibleCrates(frontier, newFrontier, accessibleCrates);
933
934         packingOrder.add(crate);
935         hasChanged = true;
936     } else {
937         crate.addCrate();
938     }
939
940     inRoom.removeCrate();
941 }
942
943 if (!hasChanged) {
944     for (TileInfo t : targets) {
945         t.removeCrate();
946     }
947
948     return false;
949 }
950 }
951
952 for (TileInfo t : targets) {
953     t.removeCrate();
954 }
955
956 Collections.reverse(packingOrder);
957 room.setPackingOrder(packingOrder);
958
959 return true;
960 }
961
962 /**
963  * Find accessible crates using bfs from lastFrontier.
964  *
965  * @param lastFrontier starting point of the bfs
966  * @param newFrontier a non-null list that will contain the next tile info to visit
967  * @param out a list that will contain accessible crates
968  */
969
970 private void findAccessibleCrates(List<TileInfo> lastFrontier, List<TileInfo>
971 ↪ newFrontier, List<TileInfo> out) {
972     newFrontier.clear();
973
974     for (int i = 0; i < lastFrontier.size(); i++) {
975         TileInfo tile = lastFrontier.get(i);
976
977         if (!tile.isMarked()) {
978             tile.mark();
979             if (tile.anyCrate()) {
980                 out.add(tile);
981             } else {
982                 for (Direction dir : Direction.VALUES) {
983                     TileInfo adj = tile.adjacent(dir);
984
985                     if (!adj.isMarked() && !adj.isWall()) {
986                         newFrontier.add(adj);
987                     }
988                 }
989             }
990         }
991     }
992 }

```



```

990     }
991
992     if (!newFrontier.isEmpty()) {
993         findAccessibleCrates(newFrontier, lastFrontier, out);
994     } else {
995         lastFrontier.clear();
996     }
997 }
998
999 private void computeTileToTargetsDistances() {
1000
1001     List<Integer> targetIndices = new ArrayList<>();
1002
1003     targetCount = 0;
1004     for (int y = 0; y < height; y++) {
1005         for (int x = 0; x < width; x++) {
1006             if (this.content[y][x].isTarget() || this.content[y][x].isCrateOnTarget())
1007                 ↪ {
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
1023                 ↪ TileInfo.TargetRemoteness[targetIndices.size()]);
1024
1025             for (int j = 0; j < targetIndices.size(); j++) {
1026
1027                 final int targetIndex = targetIndices.get(j);
1028                 final int d = (t.isFloor() || t.isTarget()
1029                     ? playerAStar.findPath(t, getAt(targetIndex), null,
1030                     ↪ null).getDist()
1031                     : 0);
1032
1033                 if (d < minDistToTarget) {
1034                     minDistToTarget = d;
1035                     minDistToTargetIndex = j;
1036                 }
1037
1038                 getAt(x, y).getTargets()[j] = new
1039                     ↪ TileInfo.TargetRemoteness(targetIndex, d);
1040             }
1041             Arrays.sort(getAt(x, y).getTargets());
1042             getAt(x, y).setNearestTarget(new
1043                 ↪ TileInfo.TargetRemoteness(minDistToTargetIndex, minDistToTarget));
1044         }
1045     }
1046 }

```

```

1047 // * DYNAMIC *
1048
1049
1050 /**
1051  * Find reachable tiles
1052  * @param playerPos The indic of the case on which the player currently is.
1053  */
1054 public void findReachableCases(int playerPos) {
1055     findReachableCases(getAt(playerPos));
1056 }
1057
1058 public void findReachableCases(TileInfo tile) {
1059     reachableMarkSystem.unmarkAll();
1060     findReachableCases_aux(tile);
1061 }
1062
1063 private void findReachableCases_aux(TileInfo tile) {
1064     tile.setReachable(true);
1065     for (Direction d : Direction.VALUES) {
1066         TileInfo adjacent = tile.adjacent(d);
1067
1068         // the second part of the condition avoids to check already processed cases
1069         if (!adjacent.isSolid() && !adjacent.isReachable()) {
1070             findReachableCases_aux(adjacent);
1071         }
1072     }
1073 }
1074
1075
1076
1077 private int topX = 0;
1078 private int topY = 0;
1079
1080 /**
1081  * This method compute the top left reachable position of the player of pushing a
1082  * crate
1083  * at (crateToMoveX, crateToMoveY) to (destX, destY). It is used to calculate the
1084  * position
1085  * of the player in a {@link State}.
1086  * This is also an example of use of {@link MarkSystem}
1087  *
1088  * @return the top left reachable position after pushing the crate
1089  * @see MarkSystem
1090  * @see Mark
1091  */
1092 @Override
1093 public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
1094     // temporary move the crate
1095     crate.removeCrate();
1096     crateDest.addCrate();
1097
1098     topX = width;
1099     topY = height;
1100
1101     markSystem.unmarkAll();
1102     topLeftReachablePosition_aux(crate);
1103
1104     // undo
1105     crate.addCrate();
1106     crateDest.removeCrate();
1107
1108     return topY * width + topX;

```

```

1107 }
1108
1109 private void topLeftReachablePosition_aux(TileInfo tile) {
1110     if (tile.getY() < topY || (tile.getY() == topY && tile.getX() < topX)) {
1111         topX = tile.getX();
1112         topY = tile.getY();
1113     }
1114
1115     tile.mark();
1116     for (Direction d : Direction.VALUES) {
1117         TileInfo adjacent = tile.adjacent(d);
1118
1119         if (!adjacent.isSolid() && !adjacent.isMarked()) {
1120             topLeftReachablePosition_aux(adjacent);
1121         }
1122     }
1123 }
1124
1125 // *****
1126 // * GETTERS / SETTERS *
1127 // *****
1128
1129 public StaticBoard staticBoard() {
1130     return staticBoard;
1131 }
1132
1133 /**
1134  * Returns the number of target i.e. tiles on which a crate has to be pushed to solve
1135  * the level on the board
1136  * @return the number of target i.e. tiles on which a crate has to be pushed to solve
1137  * the level on the board
1138  */
1139 public int getTargetCount() {
1140     return targetCount;
1141 }
1142
1143 /**
1144  * Returns all tunnels that are in this board
1145  *
1146  * @return all tunnels that are in this board
1147  */
1148 public List<Tunnel> getTunnels() {
1149     return tunnels;
1150 }
1151
1152 /**
1153  * Returns all rooms that are in this board
1154  *
1155  * @return all rooms that are in this board
1156  */
1157 public List<Room> getRooms() {
1158     return rooms;
1159 }
1160
1161 public boolean isGoalRoomLevel() {
1162     return isGoalRoomLevel;
1163 }
1164
1165 public PlayerAStar getPlayerAStar() {
1166     return playerAStar;

```

```

1167     }
1168
1169     public CrateAStar getCrateAStar() {
1170         return crateAStar;
1171     }
1172
1173     public CratePlayerAStar getCratePlayerAStar() {
1174         return cratePlayerAStar;
1175     }
1176
1177     @Override
1178     public Corral getCorral(TileInfo tile) {
1179         return corralDetector.findCorral(tile);
1180     }
1181
1182     @Override
1183     public CorralDetector getCorralDetector() {
1184         return corralDetector;
1185     }
1186
1187     /**
1188     ↪ * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking
1189     ↪ 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
1200     ↪ #findReachableCases(int)} algorithm
1201     ↪ *
1202     ↪ * @return the reachable mark system
1203     ↪ * @see MarkSystem
1204     ↪ */
1205     public MarkSystem getReachableMarkSystem() {
1206         return reachableMarkSystem;
1207     }
1208
1209     /**
1210     ↪ * Creates a {@linkplain MarkSystem mark system} that apply the specified reset
1211     ↪ * consumer to every <strong>non-wall</strong> {@linkplain TileInfo tile info}
1212     ↪ * that are in this {@linkplain Board board}.
1213     ↪ *
1214     ↪ * @param reset the reset function
1215     ↪ * @return a new MarkSystem
1216     ↪ * @see MarkSystem
1217     ↪ * @see Mark
1218     ↪ */
1219     private MarkSystem newMarkSystem(Consumer<TileInfo> reset) {
1220         return new AbstractMarkSystem() {
1221             @Override
1222             public void reset() {
1223                 mark = 0;
1224                 forEachNotWall(reset);
1225             }
1226         };
1227     }

```

```

1227 protected class StaticBoard extends GenericBoard {
1228
1229     private final List<ImmutableTunnel> tunnels;
1230     private final List<ImmutableRoom> rooms;
1231
1232     public StaticBoard() {
1233         super(MutableBoard.this.width, MutableBoard.this.height);
1234
1235         StaticTile[] [] content = new StaticTile[height][width];
1236         this.content = content;
1237
1238         for (int y = 0; y < height; y++) {
1239             for (int x = 0; x < width; x++) {
1240                 content[y][x] = new StaticTile(this, MutableBoard.this.content[y][x]);
1241             }
1242         }
1243
1244         tunnels = MutableBoard.this.tunnels.stream()
1245             .map((t) -> new ImmutableTunnel(this, t)).toList();
1246         rooms = MutableBoard.this.rooms.stream()
1247             .map((r) -> new ImmutableRoom(this, r)).toList();
1248
1249         linkTunnelsRoomsAndTileInfos(content);
1250     }
1251
1252     private void linkTunnelsRoomsAndTileInfos(StaticTile[] [] content) {
1253         Map<Room, ImmutableRoom> roomMap = new HashMap<>(rooms.size());
1254         for (int i = 0; i < rooms.size(); i++) {
1255             roomMap.put(MutableBoard.this.rooms.get(i), rooms.get(i));
1256         }
1257
1258         Map<Tunnel, ImmutableTunnel> tunnelMap = new HashMap<>(tunnels.size());
1259         for (int i = 0; i < tunnels.size(); i++) {
1260             tunnelMap.put(MutableBoard.this.tunnels.get(i), tunnels.get(i));
1261         }
1262
1263         // add rooms to tunnels
1264         List<Tunnel> originalTunnel = MutableBoard.this.tunnels;
1265         for (int i = 0; i < tunnels.size(); i++) {
1266             ImmutableTunnel t = tunnels.get(i);
1267             if (originalTunnel.get(i).rooms != null) {
1268                 t.rooms = originalTunnel.get(i).rooms.stream()
1269                     .map(r -> (Room) roomMap.get(r)).toList();
1270             }
1271         }
1272
1273         // add tunnels to rooms
1274         List<Room> originalRooms = MutableBoard.this.rooms;
1275         for (int i = 0; i < rooms.size(); i++) {
1276             ImmutableRoom r = rooms.get(i);
1277             if (originalRooms.get(i).tunnels != null) {
1278                 r.tunnels = originalRooms.get(i).tunnels.stream()
1279                     .map(t -> (Tunnel) tunnelMap.get(t)).toList();
1280             }
1281         }
1282
1283         // add tunnels, rooms to tile info
1284         for (int y = 0; y < getHeight(); y++) {
1285             for (int x = 0; x < getWidth(); x++) {
1286                 TileInfo original = MutableBoard.this.content[y][x];
1287                 StaticTile dest = content[y][x];
1288

```

```

1289         dest.tunnel = tunnelMap.get(original.getTunnel());
1290         dest.room = roomMap.get(original.getRoom());
1291
1292         if (original.getTunnelExit() != null) {
1293             dest.exit = original.getTunnelExit(); // it is immutable !
1294         }
1295     }
1296 }
1297
1298
1299 @Override
1300 public int getWidth() {
1301     return MutableBoard.this.getWidth();
1302 }
1303
1304 @Override
1305 public int getHeight() {
1306     return MutableBoard.this.getHeight();
1307 }
1308
1309 @Override
1310 public int getTargetCount() {
1311     return MutableBoard.this.getTargetCount();
1312 }
1313
1314 @SuppressWarnings("unchecked")
1315 @Override
1316 public List<Tunnel> getTunnels() {
1317     return (List<Tunnel>) ((List<?>) tunnels); // this is black magic
1318 }
1319
1320 @SuppressWarnings("unchecked")
1321 @Override
1322 public List<Room> getRooms() {
1323     return (List<Room>) ((List<?>) rooms); // more black magic !
1324 }
1325
1326 @Override
1327 public boolean isGoalRoomLevel() {
1328     return MutableBoard.this.isGoalRoomLevel();
1329 }
1330
1331 @Override
1332 public MarkSystem getMarkSystem() {
1333     return null;
1334 }
1335
1336 @Override
1337 public MarkSystem getReachableMarkSystem() {
1338     return null;
1339 }
1340 }
1341
1342 /**
1343  * A TileInfo that contains only static information
1344  */
1345 protected static class StaticTile extends GenericTileInfo {
1346
1347     private final boolean deadTile;
1348
1349     private final TargetRemoteness[] targets;
1350     private final TargetRemoteness nearestTarget;

```

```

1351
1352 private ImmutableTunnel tunnel;
1353 private ImmutableRoom room;
1354 private Tunnel.Exit exit;
1355
1356 public StaticTile(StaticBoard staticBoard, TileInfo tile) {
1357     super(staticBoard, removeCrate(tile.getTile()), tile.getX(), tile.getY());
1358     this.deadTile = tile.isDeadTile();
1359
1360     if (tile.getTargets() == null) {
1361         targets = null;
1362     } else {
1363         targets = Arrays.copyOf(tile.getTargets(), tile.getTargets().length);
1364     }
1365
1366     this.nearestTarget = tile.getNearestTarget();
1367 }
1368
1369 private static Tile removeCrate(Tile tile) {
1370     if (tile == Tile.CRATE) {
1371         return Tile.FLOOR;
1372     } else if (tile == Tile.CRATE_ON_TARGET) {
1373         return Tile.TARGET;
1374     } else {
1375         return tile;
1376     }
1377 }
1378
1379 @Override
1380 public boolean isDeadTile() {
1381     return deadTile;
1382 }
1383
1384 @Override
1385 public boolean isReachable() {
1386     return false;
1387 }
1388
1389 @Override
1390 public Tunnel getTunnel() {
1391     return tunnel;
1392 }
1393
1394 @Override
1395 public Tunnel.Exit getTunnelExit() {
1396     return exit;
1397 }
1398
1399 @Override
1400 public boolean isInATunnel() {
1401     return tunnel != null;
1402 }
1403
1404 @Override
1405 public Room getRoom() {
1406     return room;
1407 }
1408
1409 @Override
1410 public boolean isInARoom() {
1411     return room != null;
1412 }

```

```

1413
1414     @Override
1415     public boolean isMarked() {
1416         return false;
1417     }
1418
1419     @Override
1420     public int getCrateIndex() {
1421         return -1;
1422     }
1423
1424     @Override
1425     public TargetRemoteness getNearestTarget() {
1426         return nearestTarget;
1427     }
1428
1429     @Override
1430     public TargetRemoteness[] getTargets() {
1431         return targets;
1432     }
1433 }
1434
1435 private static class ImmutableTunnel extends Tunnel {
1436
1437     public ImmutableTunnel(StaticBoard board, Tunnel tunnel) {
1438         start = board.getAt(tunnel.start.getIndex());
1439         end = board.getAt(tunnel.end.getIndex());
1440
1441         if (startOut != null) {
1442             startOut = board.getAt(tunnel.startOut.getIndex());
1443         }
1444         if (endOut != null) {
1445             endOut = board.getAt(tunnel.endOut.getIndex());
1446         }
1447         playerOnlyTunnel = tunnel.isPlayerOnlyTunnel();
1448         isOneway = tunnel.isOneway();
1449     }
1450
1451     @Override
1452     public void createTunnelExits() {
1453         throw new UnsupportedOperationException();
1454     }
1455
1456     @Override
1457     public void addRoom(Room room) {
1458         throw new UnsupportedOperationException();
1459     }
1460
1461     @Override
1462     public void setStart(TileInfo start) {
1463         throw new UnsupportedOperationException();
1464     }
1465
1466     @Override
1467     public void setEnd(TileInfo end) {
1468         throw new UnsupportedOperationException();
1469     }
1470
1471     @Override
1472     public void setStartOut(TileInfo startOut) {
1473         throw new UnsupportedOperationException();
1474     }

```



```

1475
1476     @Override
1477     public void setEndOut(TileInfo endOut) {
1478         throw new UnsupportedOperationException();
1479     }
1480
1481     @Override
1482     public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
1483         throw new UnsupportedOperationException();
1484     }
1485
1486     @Override
1487     public void setCrateInside(boolean crateInside) {
1488         throw new UnsupportedOperationException();
1489     }
1490
1491     @Override
1492     public void setOneway(boolean oneway) {
1493         throw new UnsupportedOperationException();
1494     }
1495
1496     @Override
1497     public boolean crateInside() {
1498         return false;
1499     }
1500 }
1501
1502 private static class ImmutableRoom extends Room {
1503
1504     public ImmutableRoom(StaticBoard board, Room room) {
1505         goalRoom = room.isGoalRoom();
1506
1507         for (TileInfo t : room.getTiles()) {
1508             tiles.add(board.getAt(t.getIndex()));
1509         }
1510         for (TileInfo t : room.getTargets()) {
1511             targets.add(board.getAt(t.getIndex()));
1512         }
1513         if (room.getPackingOrder() != null) {
1514             packingOrder = new ArrayList<>();
1515             for (TileInfo t : room.getPackingOrder()) {
1516                 packingOrder.add(board.getAt(t.getIndex()));
1517             }
1518         }
1519     }
1520
1521     @Override
1522     public void addTunnel(Tunnel tunnel) {
1523         throw new UnsupportedOperationException();
1524     }
1525
1526     @Override
1527     public void addTile(TileInfo tile) {
1528         throw new UnsupportedOperationException();
1529     }
1530
1531     @Override
1532     public void setGoalRoom(boolean goalRoom) {
1533         throw new UnsupportedOperationException();
1534     }
1535
1536     @Override

```

```

1537     public void setPackingOrder(List<TileInfo> packingOrder) {
1538         throw new UnsupportedOperationException();
1539     }
1540
1541     @Override
1542     public void setPackingOrderIndex(int packingOrderIndex) {
1543         throw new UnsupportedOperationException();
1544     }
1545
1546     @Override
1547     public int getPackingOrderIndex() {
1548         return -1;
1549     }
1550 }
1551 }

```

Direction

```

1 package fr.valax.sokoshell.solver.board;
2
3 /**
4  * A small but super useful enumeration. Contains all direction: {@link Direction#LEFT},
5  * ↩️ {@link Direction#UP},
6  * {@link Direction#RIGHT} and {@link Direction#DOWN}.
7  *
8  * @author PoulpogGaz
9  * @author darth-mole
10 */
11 public enum Direction {
12     LEFT(-1, 0),
13     UP(0, -1),
14     RIGHT(1, 0),
15     DOWN(0, 1);
16
17     /**
18      * Directions along the horizontal axis
19      */
20     public static final Direction[] HORIZONTAL = new Direction[] {LEFT, RIGHT};
21
22     /**
23      * Directions along the vertical axis
24      */
25     public static final Direction[] VERTICAL = new Direction[] {UP, DOWN};
26
27     public static final Direction[] VALUES = new Direction[] {LEFT, UP, RIGHT, DOWN};
28
29     private final int dirX;
30     private final int dirY;
31
32     Direction(int dirX, int dirY) {
33         this.dirX = dirX;
34         this.dirY = dirY;
35     }
36
37     public int dirX() { return dirX; }
38     public int dirY() { return dirY; }
39
40     /**
41      * Rotate the rotation by 90°. For {@link Direction#UP} it returns {@link
42      * ↩️ Direction#LEFT}
43      */

```

```

43     * @return the direction rotated by 90°
44     */
45     public Direction left() {
46         return switch (this) {
47             case DOWN -> RIGHT;
48             case LEFT -> DOWN;
49             case UP -> LEFT;
50             case RIGHT -> UP;
51         };
52     }
53
54     /**
55      * Rotate the rotation by -90°. For {@link Direction#UP} it returns {@link
↪ Direction#RIGHT}
56      *
57      * @return the direction rotated by -90°
58      */
59     public Direction right() {
60         return switch (this) {
61             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#RIGHT} 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     }

```

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}
13  * → implementations.
14  * It defines all read-only methods, as well as a way to store the tiles. It is
15  * → essentially a 2D-array of
16  * * {@link TileInfo}, the indices being the y and x coordinates (i.e. {@code content[y][x]}
17  * → is the tile at (x;y)).
18  *
19  * @see Board
20  * @see TileInfo
21  */
22 public abstract class GenericBoard implements Board {
23
24     protected final int width;
25
26     protected final int height;
27
28     protected TileInfo[][] content;
29
30     public GenericBoard(int width, int height) {
31         this.width = width;
32         this.height = height;
33     }
34
35     @SuppressWarnings("CopyConstructorMissesField")
36     public GenericBoard(Board other) {
37         this(other.getWidth(), other.getHeight());
38     }
39
40     @Override
41     public int getWidth() { return width; }
42
43     @Override
44     public int getHeight() { return height; }
45
46     @Override
47     public int getY(int index) { return index / width; }
48
49     @Override
50     public int getX(int index) { return index % width; }
51
52     @Override
53     public int getIndex(int x, int y) { return y * width + x; }
54
55     @Override
56     public TileInfo getAt(int index) {
57         return content[getY(index)][getX(index)];
58     }
59 }

```

```

56
57 @Override
58 public TileInfo getAt(int x, int y) {
59     return content[y][x];
60 }
61
62
63 // SETTERS: throw UnsupportedOperationException as this object is immutable //
64 @Override
65 public void forEach(Consumer<TileInfo> consumer) {
66     throw new UnsupportedOperationException("Board is immutable");
67 }
68
69 @Override
70 public void setAt(int index, Tile tile) {
71     throw new UnsupportedOperationException("Board is immutable");
72 }
73
74 @Override
75 public void setAt(int x, int y, Tile tile) {
76     throw new UnsupportedOperationException("Board is immutable");
77 }
78
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
100 ↪ now) not to be used by solvers //
101
102 @Override
103 public void initForSolver() {
104     throw new UnsupportedOperationException("Board is not intended for solvers");
105 }
106
107 @Override
108 public void computeFloors() {
109     throw new UnsupportedOperationException("Board is not intended for solvers");
110 }
111
112 @Override
113 public void forEachNotWall(Consumer<TileInfo> consumer) {
114     throw new UnsupportedOperationException("Board is not intended for solvers");
115 }
116
117 @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
17  * width, the height etc.
18  * Implementations of this interface are meant to be used with a {@link TileInfo}
19  * implementation.
20  * This class also defines static and dynamic analysis of the Sokoban board, for instance
21  * for solving purposes.
22  * Such properties are the following:
23  * <ul>
24  *     <li>Static</li>
25  *     <ul>
26  *         <li>Dead positions: cases that make the level unsolvable when a crate is pushed
27  *         on them</li>
28  *     </ul>
29  *     <li>Dynamic</li>
30  *     <ul>
31  *         <li>Reachable cases: cases that the player can reach according to his
32  *         position</li>
33  *     </ul>
34  * </ul>
35  * @see TileInfo
36  */
37 public interface Board {
38
39     int MINIMUM_WIDTH = 5;
40     int MINIMUM_HEIGHT = 5;
41
42     // GETTERS //
43
44     /**
45      * Returns the width of the board
46      *
47      * @return the width of the board
48      */
49     int getWidth();
50
51     /**
52      * Returns the height of the board
53      *
54      * @return the height of the board
55      */
56     int getHeight();
57
58     /**
59      * Returns the number of target i.e. tiles on which a crate has to be pushed to solve
60      * the level on the board
61      *
62      * @return the number of target i.e. tiles on which a crate has to be pushed to solve
63      * the level on the board
64      */
65     int getTargetCount();
66
67     /**
68      * Convert an index to a position on the y-axis
69      *
70      * @param index the index to convert
71      * @return the converted position

```

```

66     */
67     int getY(int index);
68
69     /**
70      * Convert an index to a position on the x-axis
71      *
72      * @param index the index to convert
73      * @return the converted position
74      */
75     int getX(int index);
76
77     /**
78      * Convert a (x;y) position to an index
79      *
80      * @param x Coordinate on x-axis
81      * @param y Coordinate on y-axis
82      * @return the converted index
83      */
84     int getIndex(int x, int y);
85
86     /**
87      * Returns the {@link TileInfo} at the specific index
88      *
89      * @param index the index of the {@link TileInfo}
90      * @return the TileInfo at the specific index
91      * @throws IndexOutOfBoundsException if the index lead to a position outside the board
92      * @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
184  * as expected.</strong>
185  *
186  * @return {@code true} if the board is completed, false otherwise
187  */
188 default boolean isCompletedWith(State s) {
189     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 /**
229  * 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 // SETTERS //
245
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
399  ↪ crate
400  * at crate to crateDest. It is used to calculate the position
401  * of the player in a {@link State}.
402  * This is also an example of use of {@link MarkSystem}
403  *
404  * @return the top left reachable position after pushing the crate
405  * @see MarkSystem
406  * @see Mark
407  */
408 int topLeftReachablePosition(TileInfo crate, TileInfo crateDest);
409
410 /**
411  * @param tile tile
412  * @return the corral in which {@code tile} is
413  */
414 Corral getCorral(TileInfo tile);
415
416 /**
417  * @return the {@link CorralDetector} used to find corrals
418  */
419 CorralDetector getCorralDetector();
420 }

```

State

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.valax.sokoshell.utils.SizeOf;
4
5 import java.util.Arrays;
6 import java.util.Random;

```

```

7
8 /**
9  * A state represents an arrangement of the crates in the board and the location of the
10 ↪ player.
11  *
12  * @implNote <strong>DO NOT MODIFY THE ARRAY AFTER THE INITIALIZATION. THE HASH WON'T BE
13 ↪ RECALCULATED</strong>
14  * @author darth-mole
15  * @author PoulpoGaz
16  */
17 public class State {
18
19     // http://sokobano.de/wiki/index.php?title=Solver#Hash_Function
20     // https://en.wikipedia.org/wiki/Zobrist_hashing
21     protected static int[][] zobristValues;
22
23     /**
24     * @param minSize minSize is the number of tile in the board
25     */
26     public static void initZobristValues(int minSize) {
27         int i;
28         if (zobristValues == null) {
29             i = 0;
30             zobristValues = new int[minSize][2];
31         } else if (zobristValues.length < minSize) {
32             i = zobristValues.length;
33             zobristValues = Arrays.copyOf(zobristValues, minSize);
34         } else {
35             i = zobristValues.length;
36         }
37
38         Random random = new Random();
39         for (; i < zobristValues.length; i++) {
40             if (zobristValues[i] == null) {
41                 zobristValues[i] = new int[2];
42             }
43
44             zobristValues[i][0] = random.nextInt();
45             zobristValues[i][1] = random.nextInt();
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] ^
76      ↪   zobristValues[newPlayerPos][0] // 'moves' the player in the hash
77      ↪   ^ zobristValues[newCrates[crateToMove]][1] ^
78      ↪   zobristValues[crateDestination][1]; // 'moves' the crate in the hash
79      newCrates[crateToMove] = crateDestination;
80
81      return new State(newPlayerPos, newCrates, hash, this);
82  }
83
84  public long approxSizeOfAccurate() {
85      return SizeOf.getStateLayout().instanceSize() +
86      ↪   SizeOf.getIntArrayLayout().instanceSize() +
87      ↪   (long) Integer.BYTES * cratesIndices.length;
88  }
89
90  public long approxSizeOf() {
91      return 32 +
92      ↪   16 +
93      ↪   (long) Integer.BYTES * cratesIndices.length;
94  }
95
96  /**
97   * The index of the case of the board on which the player is.
98   */
99  public int playerPos() {
100      return playerPos;
101  }
102
103  /**
104   * The index of the cases of the board on which the crates are.
105   */
106  public int[] cratesIndices() {
107      return cratesIndices;
108  }
109
110  public int hash() {
111      return hash;
112  }
113
114  /**
115   * The state in which the board was before coming to this state.
116   */
117  public State parent() {
118      return parent;
119  }
120
121  @Override
122  public boolean equals(Object o) {
123      if (this == o) return true;
124      if (o == null || getClass() != o.getClass()) return false;
125
126      State state = (State) o;

```

```

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) {
169             hash ^= zobristValues[crate][1];
170         }
171
172         return hash;
173     }
174
175     @Override
176     public String toString() {
177         StringBuilder sb = new StringBuilder();
178         sb.append("Player: ").append(playerPos).append(", Crates: [");
179
180         for (int i = 0; i < cratesIndices.length; i++) {
181             int crate = cratesIndices[i];
182             sb.append(crate);
183
184             if (i + 1 < cratesIndices.length) {
185                 sb.append("; ");
186             }
187         }
188

```



```

189         sb.append("], hash: ").append(hash);
190
191         return sb.toString();
192     }
193 }

```

ReachableTiles

```

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.mark.FixedSizeMarkSystem;
6 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
7
8 public class ReachableTiles {
9
10     protected final FixedSizeMarkSystem reachable;
11
12     public ReachableTiles(Board board) {
13         reachable = new FixedSizeMarkSystem(board.getWidth() * board.getHeight());
14     }
15
16     public boolean isReachable(TileInfo tile) {
17         return reachable.isMarked(tile.getIndex());
18     }
19
20     public void findReachableCases(TileInfo origin) {
21         reachable.unmarkAll();
22         findReachableCases_aux(origin);
23     }
24
25     private void findReachableCases_aux(TileInfo tile) {
26         reachable.mark(tile.getIndex());
27         for (Direction d : Direction.VALUES) {
28             TileInfo adjacent = tile.adjacent(d);
29
30             // the second part of the condition avoids to check already processed cases
31             if (!adjacent.isSolid() && !isReachable(adjacent)) {
32                 findReachableCases_aux(adjacent);
33             }
34         }
35     }
36 }

```

Solver

```

1 package fr.valax.sokoshell.solver;
2
3 import java.util.List;
4
5 /**
6  * Defines the basics for all sokoban solver
7  *
8  * @author darth-mole
9  * @author PoulpoGaz
10  */
11 public interface Solver {
12
13     String DFS = "DFS";
14     String BFS = "BFS";

```

```

15     String A_STAR = "A*";
16
17     /**
18      * Try to solve the sokoban that is in the {@link SolverParameters}.
19      * @param params non null solver parameters
20      * @return a solution object
21      * @see SolverReport
22      * @see SolverParameters
23      */
24     SolverReport solve(SolverParameters params);
25
26     /**
27      * @return the name of solver
28      */
29     String getName();
30
31     /**
32      * @return {@code true} if the solver is running
33      */
34     boolean isRunning();
35
36     /**
37      * Try to stop the solver if it is running.
38      * When the solver is not running, it does nothing and returns {@code false}.
39      * A solver that doesn't support stopping must return {@code false}
40      * @return {@code true} if the solver was stopped, or if it registers the stop action.
41      * Otherwise, it returns {@code false}.
42      */
43     boolean stop();
44
45     /**
46      * Returns parameters accepted by this solver.
47      * The list returned is always a new one except when the solver don't have any
↪ parameter.
48      *
49      * @return Parameters accepted by this solver.
50      */
51     List<SolverParameter> getParameters();
52 }

```

DeadlockTable

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.valax.sokoshell.graphics.style.BasicStyle;
4 import fr.valax.sokoshell.readers.XSBReader;
5 import fr.valax.sokoshell.solver.board.Board;
6 import fr.valax.sokoshell.solver.board.Direction;
7 import fr.valax.sokoshell.solver.board.MutableBoard;
8 import fr.valax.sokoshell.solver.board.tiles.Tile;
9 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
10
11 import java.io.*;
12 import java.nio.file.Files;
13 import java.nio.file.Path;
14 import java.util.*;
15 import java.util.concurrent.ForkJoinPool;
16 import java.util.concurrent.RecursiveTask;
17 import java.util.concurrent.atomic.AtomicInteger;
18 import java.util.function.Function;
19
20 public class DeadlockTable {

```

```

21
22     protected static final int NOT_A_DEADLOCK = 0;
23     protected static final int MAYBE_A_DEADLOCK = 1;
24     protected static final int A_DEADLOCK = 2;
25
26     protected static final DeadlockTable DEADLOCK = new DeadlockTable(A_DEADLOCK);
27     protected static final DeadlockTable NOT_DEADLOCK = new DeadlockTable(NOT_A_DEADLOCK);
28
29     protected final int deadlock;
30
31     protected final int x; // relative to player x
32     protected final int y; // relative to player y
33     protected final DeadlockTable floorChild;
34     protected final DeadlockTable wallChild;
35     protected final DeadlockTable crateChild;
36
37     private DeadlockTable(int deadlock) {
38         this(deadlock, -1, -1, null, null, null);
39     }
40
41     public DeadlockTable(int deadlock, int x, int y,
42         DeadlockTable floorChild, DeadlockTable wallChild, DeadlockTable
43         ↪ crateChild) {
44         this.deadlock = deadlock;
45         this.x = x;
46         this.y = y;
47         this.floorChild = floorChild;
48         this.wallChild = wallChild;
49         this.crateChild = crateChild;
50     }
51
52     public boolean isDeadlock(TileInfo player, Direction pushDir) {
53         Board board = player.getBoard();
54
55         if (player.adjacent(pushDir).isCrateOnTarget()) {
56             return false;
57         }
58
59         return switch (pushDir) {
60             case LEFT -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.y,
61                 ↪ player.getY() + t.x));
62             case UP -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.x,
63                 ↪ player.getY() + t.y));
64             case RIGHT -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.y,
65                 ↪ player.getY() - t.x));
66             case DOWN -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.x,
67                 ↪ player.getY() - t.y));
68         };
69     }
70
71     private boolean isDeadlock(Function<DeadlockTable, TileInfo> getTile) {
72         if (deadlock == A_DEADLOCK) {
73             return true;
74         } else if (deadlock == NOT_A_DEADLOCK) {
75             return false;
76         }
77
78         TileInfo tile = getTile.apply(this);
79
80         if (tile == null) {
81             return false;
82         }

```

```

78         return switch (tile.getTile()) {
79             case FLOOR -> floorChild.isDeadlock(getTile);
80             case WALL -> wallChild.isDeadlock(getTile);
81             case CRATE -> crateChild.isDeadlock(getTile);
82             default -> false;
83         };
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,
164         ↪ board.getHeight() - 3);
165 }
166
167 private static int countNotDetectedDeadlock(DeadlockTable table, Board board, int
168     ↪ playerX, int playerY) {
169     if (table.deadlock == A_DEADLOCK) {
170         State state = createState(board, playerX, playerY);
171
172         // but dead tiles aren't computed...
173         if (FreezeDeadlockDetector.checkFreezeDeadlock(board, state)) {
174             return 0;
175         }
176
177         CorralDetector detector = board.getCorralDetector();
178         detector.findCorral(board, playerX, playerY);
179         detector.findPICorral(board, state.cratesIndices());
180
181         boolean deadlock = false;
182         for (Corral c : detector.getCorrals()) {
183             if (c.isDeadlock(state, true)) {
184                 deadlock = true;
185                 break;
186             }
187         }
188
189         if (deadlock) {
190             return 0;
191         } else {
192             BasicStyle.XSB_STYLE.print(board, playerX, playerY);
193
194             return 1; // not detected !
195         }
196     } else if (table.deadlock == MAYBE_A_DEADLOCK) {
197         int n = countNotDetectedDeadlock(table.floorChild, board, playerX, playerY);
198
199         board.setAt(playerX + table.x, playerY + table.y, Tile.WALL);
200         n += countNotDetectedDeadlock(table.wallChild, board, playerX, playerY);

```

```

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,
226         ↪ board.getHeight() - 3);
227 }
228
229 public static DeadlockTable generate2(int size, int nThread) {
230     Board board = createBoard(size);
231
232     board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
233
234     ForkJoinPool pool = new ForkJoinPool(nThread <= 0 ?
235         ↪ Runtime.getRuntime().availableProcessors() : nThread);
236     GenerateDeadlockTableTask task = new GenerateDeadlockTableTask(board,
237         ↪ createOrder(size), 0, board.getWidth() / 2, board.getHeight() - 3, false);
238
239     DeadlockTable table = pool.invoke(task);
240     pool.shutdown();
241
242     return table;
243 }
244
245 private static DeadlockTable generate(Board board, int[][] order, int index, int
246     ↪ playerX, int playerY) {
247     // BasicStyle.XSB_STYLE.print(board, playerX, playerY);
248
249     if (isDeadlock_(board, playerX, playerY)) {
250         return DEADLOCK;
251     } else if (index < order.length) {
252         int relativeX = order[index][0];
253         int relativeY = order[index][1];
254
255         board.setAt(playerX + relativeX, playerY + relativeY, Tile.WALL);
256         DeadlockTable wallChild = generate(board, order, index + 1, playerX, playerY);
257
258         board.setAt(playerX + relativeX, playerY + relativeY, Tile.CRATE);
259         DeadlockTable crateChild = generate(board, order, index + 1, playerX,
260             ↪ playerY);

```

```

257         board.setAt(playerX + relativeX, playerY + relativeY, Tile.FLOOR);
258         if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
259             return NOT_DEADLOCK;
260         }
261     }
262
263     DeadlockTable floorChild = generate(board, order, index + 1, playerX,
264         ↪ playerY);
265
266     return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild,
267         ↪ wallChild, crateChild);
268 } else {
269     return NOT_DEADLOCK;
270 }
271
272 private static Board createBoard(int size) {
273     Board board = new MutableBoard(size + 4, size + 4);
274     State.initZobristValues(board.getWidth() * board.getHeight());
275
276     for (int x = 0; x < board.getWidth(); x++) {
277         board.setAt(x, 0, Tile.WALL);
278         board.setAt(x, board.getHeight() - 1, Tile.WALL);
279     }
280
281     for (int y = 0; y < board.getHeight(); y++) {
282         board.setAt(0, y, Tile.WALL);
283         board.setAt(board.getWidth() - 1, y, Tile.WALL);
284     }
285
286     return board;
287 }
288
289 protected static int[][] createOrder(int size) {
290     int[][] order = new int[size * size - 2][2];
291
292     boolean odd = size % 2 == 1;
293     int i = 0;
294     int half = size / 2;
295     for (int y = 0; y > -size; y--) {
296         for (int x = -half; x < half || (x == half && odd); x++) {
297             if (x == 0 && (y == 0 || y == -1)) {
298                 continue;
299             }
300
301             order[i] = new int[] {x, y};
302             i++;
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         return new GenerateDeadlockTableTask(board, order, index + 1, playerX,
377         ↪ 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>
418                                         ↪ toVisit) {
419     for (int i = 0; i < parent.cratesIndices().length; i++) {
420         TileInfo crate = board.getAt(parent.cratesIndices()[i]);
421
422         for (Direction dir : Direction.VALUES) {
423             TileInfo player = crate.adjacent(dir.negate());
424
425             if (!reachableTiles.isReachable(player)) {
426                 continue;
427             }
428
429             TileInfo dest = crate.adjacent(dir);
430             if (dest.isSolid()) {
431                 continue;
432             }
433
434             State child;
435             if (dest.getY() == 1 || dest.getX() == 1 || dest.getX() ==
436             ↪ board.getWidth() - 2) {

```

```

435         // remove the crate, it is outside the pattern
436         if (parent.cratesIndices().length == 1) {
437             return false; // all crates were moved outside the pattern. not a
438                             ↪ deadlock...
439         }
440
441         int topLeft = board.topLeftReachablePosition(crate, board.getAt(0,
442                             ↪ 0));
443
444         child = new State(topLeft,
445                             ↪ copyRemoveOneElement(parent.cratesIndices(), i), parent);
446
447         } else {
448             int topLeft = board.topLeftReachablePosition(crate, dest);
449             child = parent.child(topLeft, i, dest.getIndex());
450         }
451
452         if (visited.add(child)) {
453             toVisit.add(child);
454         }
455     }
456
457     return true; // not a deadlock
458 }
459
460 private static int[] copyRemoveOneElement(int[] array, int indexToRemove) {
461     int[] newArray = new int[array.length - 1];
462
463     int offset = 0;
464     for (int i = 0; i < array.length; i++) {
465         if (indexToRemove == i) {
466             offset = 1;
467         } else {
468             newArray[i - offset] = array[i];
469         }
470     }
471
472     return newArray;
473 }
474
475 private static State createState(Board board, int playerX, int playerY) {
476     List<Integer> ints = new ArrayList<>();
477
478     board.forEach(t -> {
479         if (t.anyCrate()) {
480             ints.add(t.getIndex());
481         }
482     });
483
484     return new State(playerY * board.getWidth() + playerX, ints.stream().mapToInt(i ->
485                             ↪ i).toArray(), null);
486 }
487
488 }

```

AStarSolver

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.poulpogaz.json.IJsonReader;
4 import fr.poulpogaz.json.IJsonWriter;
5 import fr.poulpogaz.json.JsonException;

```

```

6 import fr.valax.sokoshell.commands.AbstractCommand;
7 import fr.valax.sokoshell.solver.board.Direction;
8 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
9 import fr.valax.sokoshell.solver.collections.SolverPriorityQueue;
10 import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
11 import fr.valax.sokoshell.solver.heuristic.Heuristic;
12 import fr.valax.sokoshell.solver.heuristic.SimpleHeuristic;
13 import org.jline.reader.Candidate;
14 import org.jline.reader.LineReader;
15
16 import java.io.IOException;
17 import java.util.List;
18
19 public class AStarSolver extends AbstractSolver<WeightedState> {
20
21     private Heuristic heuristic;
22     private int lowerBound;
23
24     public AStarSolver() {
25         super(A_STAR);
26     }
27
28     @Override
29     protected void init(SolverParameters parameters) {
30         String heuristicName = parameters.getArgument("heuristic");
31
32         if (heuristicName.equalsIgnoreCase("simple")) {
33             heuristic = new SimpleHeuristic(board);
34         } else {
35             heuristic = new GreedyHeuristic(board);
36         }
37
38         toProcess = new SolverPriorityQueue();
39     }
40
41     @Override
42     protected void addInitialState(Level level) {
43         final State s = level.getInitialState();
44         lowerBound = heuristic.compute(s);
45
46         toProcess.addState(new WeightedState(s, 0, lowerBound));
47     }
48
49     @Override
50     protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
51         if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
52             return;
53         }
54
55         final int i = board.topLeftReachablePosition(crate, crateDest);
56         // The new player position is the crate position
57         WeightedState s = toProcess.cachedState().child(i, crate.getCrateIndex(),
58             ↪ crateDest.getIndex());
59         s.setHeuristic(heuristic.compute(s));
60
61         if (processed.add(s)) {
62             toProcess.addState(s);
63         }
64     }
65
66     @Override
67     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") ||
87             ↪ argument.equalsIgnoreCase("simple")) {
88             this.value = argument;
89         } else {
90             throw new AbstractCommand.InvalidArgument("No such heuristic: " +
91                 ↪ argument);
92         }
93     }
94
95     @Override
96     public Object get() {
97         return value;
98     }
99
100     @Override
101     public Object getDefaultValue() {
102         return "greedy";
103     }
104
105     @Override
106     public void toJson(IJsonWriter jw) throws JSONException, IOException {
107         jw.value(value);
108     }
109
110     @Override
111     public void fromJson(IJsonReader jr) throws JSONException, IOException {
112         value = jr.nextString();
113     }
114
115     @Override
116     public void complete(LineReader reader, String argument, List<Candidate>
117         ↪ candidates) {
118         candidates.add(new Candidate("simple"));
119         candidates.add(new Candidate("greedy"));
120     }
121 }

```

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;
6 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
7
8 import java.util.*;
9
10 public class Corral {
11
12     public static final int POTENTIAL_PI_CORRAL = 0;
13     public static final int IS_A_PI_CORRAL = 1;
14     public static final int NOT_A_PI_CORRAL = 2;
15
16     protected final int id;
17     protected final Board board;
18
19     protected int topX;
20     protected int topY;
21
22     protected final Set<Corral> adjacentCorrals = new HashSet<>();
23
24     /**
25      * All crates that are inside the corral and surrounding the corral
26      */
27     protected final List<TileInfo> barrier = new ArrayList<>();
28     protected final List<TileInfo> crates = new ArrayList<>();
29     protected boolean containsPlayer;
30     protected boolean adjacentToPlayerCorral; // the player corral is adjacent to itself
31     protected int isPICorral;
32     protected boolean onlyCrateOnTarget; // true if all crates in crates list are crate on
33     ↪ target
34     protected boolean isValid = false;
35
36     protected final Set<CorralState> visited = new HashSet<>();
37     protected final Queue<CorralState> toVisit = new ArrayDeque<>();
38     protected final ReachableTiles reachable;
39     protected CorralState currentState;
40     protected DeadlockTable deadlockTable;
41
42     public Corral(int id, Board board) {
43         this.id = id;
44         this.board = board;
45         this.reachable = new ReachableTiles(board);
46     }
47
48     public boolean isDeadlock(State originalState) {
49         return isDeadlock(originalState, false);
50     }
51
52     public boolean isDeadlock(State originalState, boolean forceContainsAllCrate) {
53         if (!isPICorral() ||
54             onlyCrateOnTarget ||
55             !forceContainsAllCrate && crates.size() ==
56             ↪ originalState.cratesIndices().length) {
57             return false;
58         }
59         addFrozenCrates(originalState);
60         if (!forceContainsAllCrate && crates.size() ==
61             ↪ originalState.cratesIndices().length) {
62             return false;
63         }

```

```

63
64     boolean deadlock = true;
65     CorralState firstState = removeOutsideCrate(originalState);
66
67     visited.add(firstState);
68     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
157         // ↪ tunnel.endOut
158         // this part of the code assume that there is no other crate in the tunnel.
159         // normally, this is impossible...
160
161         for (Direction pushDir : Direction.VALUES) {
162             TileInfo player = crate.adjacent(pushDir.negate());
163
164             if (reachable.isReachable(player)) {
165                 TileInfo dest = crate.getTunnelExit().getExit(pushDir);
166
167                 if (dest != null && !dest.isSolid()) {
168                     if (!addState(crateIndex, crate, dest, pushDir)) {
169                         return false; // not a deadlock
170                     }
171                 }
172             }
173         }
174
175         return true;
176     }
177
178     protected boolean addChildrenStatesDefault(int crateIndex, TileInfo crate) {
179         for (Direction d : Direction.VALUES) {
180             TileInfo crateDest = crate.adjacent(d);
181             if (crateDest.isSolid()) {
182                 continue; // The destination case is not empty
183             }
184
185             if (crateDest.isDeadTile()) {
186                 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
202             ↪ deadlock
203             continue;
204         }
205
206         // ie the crate can't be pushed to the other extremities of the tunnel
207         // however, sometimes (boxxle 24) it is useful to push the crate inside
208         // the tunnel. That's why the second addState is done (after this if)
209         // and only if this tunnel isn't oneway
210         if (!tunnel.isPlayerOnlyTunnel()) {
211             TileInfo newDest = null;
212             Direction pushDir = null;
213
214             if (crate == tunnel.getStartOut()) {
215                 if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate())
216                     ↪ {
217                         newDest = tunnel.getEndOut();
218                         pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
219                     }
220             } else {
221                 if (tunnel.getStartOut() != null &&
222                     ↪ !tunnel.getStartOut().anyCrate()) {
223                     newDest = tunnel.getStartOut();
224                     pushDir = tunnel.getStart().direction(tunnel.getStartOut());
225                 }
226             }
227
228             if (newDest != null && !newDest.isDeadTile()) {
229                 if (!addState(crateIndex, crate, newDest, pushDir)) {
230                     return false;
231                 }
232             }
233         }
234
235         if (tunnel.isOneway()) {
236             continue;
237         }
238     }
239
240     if (!addState(crateIndex, crate, crateDest, d)) {
241         return false;
242     }
243
244     return true;

```



```

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             ↪ {
399             ↪     return new CorralState(super.child(newPlayerPos, crateToMove,
400             ↪         ↪     crateDestination));
401         }
402
403         public void increaseNumberOnTarget() {
404             numOnTarget++;
405         }
406
407         public void decreaseNumberOnTarget() {
408             numOnTarget--;
409         }
410
411         public int getNumOnTarget() {
412             return numOnTarget;
413         }
414
415         public void setNumOnTarget(int numOnTarget) {
416             this.numOnTarget = numOnTarget;
417         }
418     }
419 }

```

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
11  * the same -- the only
12  * difference being in the order in which they treat the states (LIFO for DFS and FIFO for
13  * BFS).
14  */
15 public abstract class BruteforceSolver extends AbstractSolver<State> {
16
17     public BruteforceSolver(String name) {
18         super(name);
19     }
20
21     public static DFSSolver newDFSSolver() {
22         return new DFSSolver();
23     }
24
25     public static BFSSolver newBFSSolver() {
26         return new BFSSolver();
27     }
28
29     @Override
30     protected void addInitialState(Level level) {
31         toProcess.addState(level.getInitialState());
32     }
33
34     @Override
35     protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
36         if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
37             return;
38         }
39
40         final int i = board.topLeftReachablePosition(crate, crateDest);
41         // The new player position is the crate position
42         State s = toProcess.cachedState().child(i, crate.getCrateIndex(),
43             ↪ crateDest.getIndex());
44
45         if (processed.add(s)) {
46             toProcess.addState(s);
47         }
48     }
49
50     @Override
51     public int lowerBound() {
52         return -1;
53     }
54
55     /**
56     * Base class for DFS and BFS solvers collection (both of them use {@link
57     * ↪ ArrayDeque}), the only difference being in
58     * * which side of the queue is used (end => FIFO => DFS, start => LIFO => BFS)
59     */
60     private static abstract class BasicBruteforceSolverCollection implements
61         ↪ SolverCollection<State> {
62
63         protected final ArrayDeque<State> collection = new ArrayDeque<>();
64
65         protected State cachedState;
66
67         @Override
68         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

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.valax.sokoshell.DefaultTracker;
4
5 /**
6  * A tracker is an object that watch a {@link Trackable} and gather solver statistics
7  * @see DefaultTracker
8  * @see Trackable
9  */
10 public interface Tracker {
11
12     /**
13      * The name of the parameter
14      * @see SolverParameters
15      */
16     String TRACKER_PARAM = "tracker";
17
18     /**
19      * Get data from a {@link Trackable}
20      * @param trackable a trackable from which we get data
21      * @see Trackable
22      */
23     void updateStatistics(Trackable trackable);
24
25     /**
26      * Clear all previously gathered statistics
27      */
28     void reset();
29
30     /**
31      * Build a {@link ISolverStatistics} object. It uses the Trackable to get the last
32      ↪ data.
33      * It is called once at the end of research.
34      * @param trackable a trackable from which we get data
35      * @return solver statistics
36      * @see ISolverStatistics
37      */
38     ISolverStatistics getStatistics(Trackable trackable);
39 }

```

AbstractSolver

```
1 package fr.valax.sokoshell.solver;
2
3 import fr.valax.sokoshell.graphics.style.BasicStyle;
4 import fr.valax.sokoshell.solver.board.*;
5 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6 import fr.valax.sokoshell.solver.collections.SolverCollection;
7 import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
8 import fr.valax.sokoshell.utils.SizeOf;
9
10 import java.io.IOException;
11 import java.nio.file.Path;
12 import java.util.*;
13
14 /**
15  * This class is the base for brute-force-based solvers, i.e. solvers that use an
16  * ↪ exhaustive search to try and find a
17  * solution.
18  * @author darth-mole
19  */
20 public abstract class AbstractSolver<S> extends State implements Trackable, Solver {
21
22     protected static final String TIMEOUT = "timeout";
23     protected static final String MAX_RAM = "max-ram";
24     protected static final String ACCURATE = "accurate";
25
26     protected final String name;
27
28     protected final DeadlockTable table;
29
30     protected SolverCollection<S> toProcess;
31     protected final Set<State> processed = new HashSet<>();
32
33     protected MutableBoard board;
34
35     private boolean running = false;
36     private boolean stopped = false;
37
38     // statistics
39     private long timeStart = -1;
40     private long timeEnd = -1;
41     private int nStateProcessed = -1;
42     private int queueSize = -1;
43     private Tracker tracker;
44
45     public AbstractSolver(String name) {
46         this.name = name;
47
48         try {
49             table = DeadlockTable.read(Path.of("4x4.table"));
50         } catch (IOException e) {
51             throw new RuntimeException(e);
52         }
53     }
54
55     @Override
56     public SolverReport solve(SolverParameters params) {
57         Objects.requireNonNull(params);
58
59         // init statistics, timeout and stop
60         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
83     Level level = params.getLevel();
84
85     State.initZobristValues(level.getWidth() * level.getHeight());
86
87     final State initialState = level.getInitialState();
88     State finalState = null;
89
90     board = new MutableBoard(level);
91     board.removeStateCrates(initialState);
92     board.initForSolver();
93     board.getCorralDetector().setDeadlockTable(table);
94
95     init(params);
96     processed.clear();
97
98     addInitialState(level);
99
100    if (level.getPack().name().equals("XSokoban_90") && level.getIndex() == 3) {
101        board.getAt(9, 10).setDeadTile(true);
102    }
103
104    while (!toProcess.isEmpty() && !stopped) {
105        if (hasTimedOut(timeout)) {
106            endStatus = SolverReport.TIMEOUT;
107            break;
108        }
109
110        if (hasRamExceeded(maxRam, accurate)) {
111            endStatus = SolverReport.RAM_EXCEED;
112            break;
113        }
114
115        S state = toProcess.peekAndCacheState();
116        board.addStateCrates(state);
117
118        if (board.isCompletedWith(state)) {
119            finalState = state;
120            break;
121        }

```



```

122
123     int playerX = board.getX(state.playerPos());
124     int playerY = board.getY(state.playerPos());
125
126     CorralDetector detector = board.getCorralDetector();
127     detector.findCorral(board, playerX, playerY);
128
129     if (checkPICorralDeadlock(state)) {
130         board.removeStateCrates(state);
131         continue;
132     }
133
134     // compute after checking for corral deadlock, as corral deadlock deals with
135     ↪ tunnels
136     board.computeTunnelStatus(state);
137     board.computePackingOrderProgress(state);
138
139     addChildrenStates(board.getAt(playerX, playerY));
140     board.removeStateCrates(state);
141 }
142
143 // END OF RESEARCH
144
145 timeEnd = System.currentTimeMillis();
146 nStateProcessed = processed.size();
147 queueSize = toProcess.size();
148
149 // 'free' ram
150 processed.clear();
151 toProcess.clear();
152 board = null;
153
154 running = false;
155
156 System.out.println("END: " + finalState + " - " + endStatus);
157
158 if (endStatus != null) {
159     return SolverReport.withoutSolution(params, getStatistics(), endStatus);
160 } else if (stopped) {
161     return SolverReport.withoutSolution(params, getStatistics(),
162     ↪ SolverReport.STOPPED);
163 } else if (finalState != null) {
164     return SolverReport.withSolution(finalState, params, getStatistics());
165 } else {
166     return SolverReport.withoutSolution(params, getStatistics(),
167     ↪ SolverReport.NO_SOLUTION);
168 }
169 }
170
171 /**
172  * Initialize the solver. This method is called after the initialization of
173  * the board
174  */
175
176 protected abstract void init(SolverParameters parameters);
177
178 protected abstract void addInitialState(Level level);
179
180 protected boolean checkPICorralDeadlock(State state) {
181     CorralDetector detector = board.getCorralDetector();
182     detector.findPICorral(board, state.cratesIndices());
183
184     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
243 ↪ tunnel.endOut
244 // this part of the code assume that there is no other crate in the tunnel.
245 // normally, this is impossible...
246
247 for (Direction pushDir : Direction.VALUES) {
248     TileInfo player = crate.adjacent(pushDir.negate());
249
250     if (player.isReachable()) {
251         TileInfo dest = crate.getTunnelExit().getExit(pushDir);
252
253         if (dest != null && !dest.isSolid()) {
254             addStateCheckForGoalMacro(crate, dest, pushDir);
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         // check for tunnel
279         Tunnel tunnel = crateDest.getTunnel();
280
281         // the crate will be pushed inside the tunnel
282         if (tunnel != null) {
283             if (tunnel.crateInside()) { // pushing inside will lead to a corral
284                 ↪ 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                         ↪ {
299                             newDest = tunnel.getEndOut();
300                             pushDir = tunnel.getEnd().direction(tunnel.getEndOut());

```

```

301         } else {
302             if (tunnel.getStartOut() != null &&
303                 ↪ !tunnel.getStartOut().anyCrate()) {
304                 newDest = tunnel.getStartOut();
305                 pushDir = tunnel.getStart().direction(tunnel.getStartOut());
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
323 ↪ pushDir) {
324     Room room = dest.getRoom();
325     if (room != null && board.isGoalRoomLevel() && room.getPackingOrderIndex() >= 0) {
326         // goal macro!
327         TileInfo newDest = room.getPackingOrder().get(room.getPackingOrderIndex());
328
329         addState(crate, newDest, null);
330     } else {
331         addState(crate, dest, pushDir);
332     }
333 }
334
335 /**
336  * Check if the move leads to a deadlock.
337  * Only for simple deadlock that don't require
338  * lots of computation like PI Corral deadlock
339  *
340  * @param crate crate to move
341  * @param crateDest crate destination
342  * @param pushDir push dir of the player. If the move is a macro move,
343  * it is the last push done by the player. It can be null
344  * @return true if deadlock
345  */
346 protected boolean checkDeadlockBeforeAdding(TileInfo crate, TileInfo crateDest,
347 ↪ Direction pushDir) {
348     crate.removeCrate();
349     crateDest.addCrate();
350
351     boolean deadlock = FreezeDeadlockDetector.checkFreezeDeadlock(crateDest);
352
353     if (!deadlock && pushDir != null) {
354         deadlock = table.isDeadlock(crateDest.adjacent(pushDir.negate()), pushDir);
355     }
356
357     crate.addCrate();
358     crateDest.removeCrate();
359
360     return deadlock;
361 }

```

```

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",
428         ↪ -1));
429     parameters.add(new SolverParameter.RamParameter(MAX_RAM, -1));
430     parameters.add(new SolverParameter.Boolean(ACCURATE,
431         ↪ "Use a more accurate method to calculate ram usage", false));
432 }
433
434 private ISolverStatistics getStatistics() {
435     ISolverStatistics stats;
436
437     if (tracker != null) {
438         stats = Objects.requireNonNull(tracker.getStatistics(this));
439     } else {
440         stats = new ISolverStatistics.Basic(timeStart, timeEnd);
441     }
442
443     return stats;
444 }
445
446 @Override
447 public State currentState() {
448     if (toProcess != null && running) {
449         return toProcess.cachedState();
450     } else {
451         return null;
452     }
453 }
454
455 @Override
456 public Board staticBoard() {
457     if (board != null && running) {
458         return board.staticBoard();
459     } else {
460         return null;
461     }
462 }
463
464 @Override
465 public int nStateExplored() {
466     if (timeStart < 0) {
467         return -1;
468     } else if (timeEnd < 0) {
469         return processed.size();
470     } else {
471         return nStateProcessed;
472     }
473 }
474
475 @Override
476 public int currentQueueSize() {
477     if (timeStart < 0) {
478         return -1;
479     } else if (timeEnd < 0 && toProcess != null) {
480         return toProcess.size();
481     } else {
482         return queueSize;
483     }
484 }

```

```

481     }
482 }
483
484 @Override
485 public long timeStarted() {
486     return timeStart;
487 }
488
489 @Override
490 public long timeEnded() {
491     return timeEnd;
492 }
493
494 @Override
495 public void setTacker(Tracker tracker) {
496     this.tracker = tracker;
497 }
498
499 @Override
500 public Tracker getTracker() {
501     return tracker;
502 }
503
504 }

```

SolverParameter

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.poulpogaz.json.IJsonReader;
4 import fr.poulpogaz.json.IJsonWriter;
5 import fr.poulpogaz.json.JsonException;
6 import fr.valax.sokoshell.commands.AbstractCommand;
7 import org.jline.reader.Candidate;
8 import org.jline.reader.LineReader;
9
10 import java.io.IOException;
11 import java.util.List;
12 import java.util.Objects;
13 import java.util.regex.Matcher;
14 import java.util.regex.Pattern;
15
16 /**
17  * A parameter given to a {@link Solver}. A parameter has a name and a description.
18  * It is responsible for parsing arguments and give default value. Implementations
19  * can also define how to auto complete and must implements {@link #fromJson(IJsonReader)}
20  * and {@link #toJson(IJsonWriter)}
21  */
22 public abstract class SolverParameter {
23
24     protected final String name;
25     protected final String description;
26
27     public SolverParameter(String name, String description) {
28         this.name = name;
29         this.description = description;
30     }
31
32     public String getName() {
33         return name;
34     }
35

```

```

36 public String getDescription() {
37     return description;
38 }
39
40 public abstract void set(String argument) throws AbstractCommand.InvalidArgument;
41
42 public abstract Object get();
43
44 public Object getOrDefault() {
45     Object o = get();
46
47     if (o == null) {
48         o = Objects.requireNonNull(getDefaultValue());
49     }
50
51     return o;
52 }
53
54 public abstract Object getDefaultValue();
55
56 public boolean hasArgument() {
57     return get() != null;
58 }
59
60
61 public void complete(LineReader reader, String argument, List<Candidate> candidates) {
62
63 }
64
65 /**
66  * @implNote name is already written
67  */
68 public abstract void toJson(IJsonWriter jw) throws JSONException, IOException;
69
70 /**
71  * @implNote name is already read
72  */
73 public abstract void fromJson(IJsonReader jr) throws JSONException, IOException;
74
75
76
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
140     public void set(String argument) throws AbstractCommand.InvalidArgument {
141         try {
142             value = java.lang.Long.parseLong(argument);
143         } catch (NumberFormatException e) {
144             throw new AbstractCommand.InvalidArgument(e);
145         }
146     }
147
148     @Override
149     public Object get() {
150         return value;
151     }
152
153     @Override
154     public Object getDefaultValue() {
155         return defaultValue;
156     }
157
158     @Override
159     public void toJson(IJsonWriter jw) throws JSONException, IOException {

```

```

160         if (value != null) {
161             jw.value(value);
162         }
163     }
164
165     @Override
166     public void fromJson(IJsonReader jr) throws JSONException, IOException {
167         value = jr.nextLong();
168     }
169 }
170
171
172 public static class Boolean extends SolverParameter {
173
174     protected final boolean defaultValue;
175     protected java.lang.Boolean value = null;
176
177     public Boolean(String name, boolean defaultValue) {
178         this(name, null, defaultValue);
179     }
180
181     public Boolean(String name, String description, boolean defaultValue) {
182         super(name, description);
183         this.defaultValue = defaultValue;
184     }
185
186     @Override
187     public void set(String argument) throws AbstractCommand.InvalidArgument {
188         try {
189             int v = java.lang.Integer.parseInt(argument);
190
191             value = v != 0;
192         } catch (NumberFormatException e) {
193             value = java.lang.Boolean.parseBoolean(argument);
194         }
195     }
196
197     @Override
198     public Object get() {
199         return value;
200     }
201
202     @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
3 import fr.valax.sokoshell.solver.board.Board;
4
5 /**
6  * A solver that implements this interface allows
7  * other objects to get information about the current
8  * research.
9  * <br>
10 * Methods are by default non-synchronized and <strong>should not</strong>
11 * modify the state of the solver.
12 * Implementations are free to violate the first term of the contract
13 * <strong>(not the second)</strong>, but they must indicate it.
14 */
15 public interface Trackable extends Solver {
16
17     /**
18      * @return the number of state explored or -1
19      */

```

```

20     int nStateExplored();
21
22     /**
23      * Returns the size of the queue. The queue contains all
24      * states that will be processed in the future. It may return
25      * {@code -1} when the Solver doesn't have a queue, or it is
26      * impossible to get this information .
27      * @return the size of the queue or -1
28      */
29     int currentQueueSize();
30
31     /**
32      * @return lower bound from initial state
33      */
34     int lowerBound();
35
36     /**
37      * @return the time in milliseconds at which the solver was started
38      */
39     long timeStarted();
40
41     /**
42      * @return the time in milliseconds at which the solver finished the research or was
43 ↪ stopped
44      */
45     long timeEnded();
46
47     /**
48      * @return the state the solver is processing. It may return null
49      */
50     State currentState();
51
52     /**
53      * @return an immutable board that contains all static information.
54      * The board has no crate on it
55      */
56     Board staticBoard();
57
58     /**
59      * Set the {@link Tracker} that is tracking this trackable
60      * @param tracker the tracker
61      */
62     void setTacker(Tracker tracker);
63
64     /**
65      * @return the tracker that is tracking this trackable
66      */
67     Tracker getTracker();
68 }

```

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);
121             }
122         }
123     }
124
125     protected boolean isICorral(Corral corral) {
126         for (TileInfo crate : corral.barrier) {
127             for (Direction dir : Direction.VALUES) {
128                 TileInfo crateDest = crate.adjacent(dir);
129                 if (crateDest.isSolid()) {
130                     continue;
131                 }
132
133                 TileInfo player = crate.adjacent(dir.negate());
134                 if (player.isSolid()) {

```

```

135         continue;
136     }
137
138     Corral corralDest = findCorral(crateDest);
139     Corral playerCorral = findCorral(player);
140
141     if (corralDest == playerCorral) {
142         return false;
143     }
144 }
145 }
146
147 return true;
148 }
149
150 protected boolean isPICorral(Corral corral) {
151     if (!corral.adjacentToPlayerCorral || corral.adjacentCorrals.size() != 1) {
152         return false;
153     }
154
155     for (TileInfo crate : corral.barrier) {
156         for (Direction dir : Direction.VALUES) {
157             TileInfo crateDest = crate.adjacent(dir);
158             if (crateDest.isSolid()) {
159                 continue;
160             }
161
162             TileInfo player = crate.adjacent(dir.negate());
163             if (player.isWall()) {
164                 continue;
165             } else if (player.anyCrate()) {
166                 /*if (!corral.crates.contains(player) &&
167                     ↪ !corral.barrier.contains(player)) {
168                     return false;
169                 }*/
170                 continue;
171             }
172
173             if (crateDest.isDeadTile()) {
174                 continue; // only consider valid moves
175             }
176
177             Corral corralDest = findCorral(crateDest);
178             Corral playerCorral = findCorral(player);
179
180             if (playerCorral.containsPlayer() && playerCorral == corralDest) {
181                 return false;
182             }
183         }
184     }
185
186     return true;
187 }
188
189 protected void mergeWithAdjacents(Board board, Corral corral) {
190     while (corral.adjacentCorrals.size() > 1) {
191         Iterator<Corral> iterator = corral.adjacentCorrals.iterator();
192         Corral adj = null;
193
194         while (iterator.hasNext()) {
195             adj = iterator.next();

```

```

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()),
217         ↪ board.getAt(b.getTopX(), b.getTopY()));
218
219     if (corral == b) {
220         b = a; // this way, we can deal with corral (before a) and b, without doing
221         ↪ disjonction.
222     }
223
224     // Merge properties. It is assumed that a and b doesn't contain the player
225     // topX, topY are already updated
226     // the set currentCorrals was also updated.
227     corral.adjacentToPlayerCorral |= b.adjacentToPlayerCorral;
228     corral.onlyCrateOnTarget &= b.onlyCrateOnTarget;
229
230     // update adjacentCorrals
231     // Add all adjacents corral of b to corral, but corral is adjacent to b,
232     // we must remove it. The remove is done before addAll because the resulting
233     // set is likely to be bigger than b one.
234     b.adjacentCorrals.remove(corral);
235     // also update adjacent of b
236     for (Corral bAdj : b.adjacentCorrals) {
237         bAdj.adjacentCorrals.remove(b);
238
239         if (bAdj != corral) {
240             bAdj.adjacentCorrals.add(corral);
241         }
242     }
243     corral.adjacentCorrals.remove(b);
244     corral.adjacentCorrals.addAll(b.adjacentCorrals);
245
246     // update barrier and crates
247     for (TileInfo tile : b.crates) {
248         if (!corral.crates.contains(tile)) {
249             corral.crates.add(tile);
250         }
251     }
252
253     // merge the two barrier. Some crates aren't in a barrier.
254     for (TileInfo tile : b.barrier) {
255         if (!corral.barrier.contains(tile)) {
256             corral.barrier.add(tile);
257         }
258     }

```



```

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
316             ↪ set is faster
317             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 = corral1;
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 = corral1;
418         Corral oldCorral = corral2;
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 &&
425             ↪ oldCorral.topX < newCorral.topX)) {
426             newCorral.topX = oldCorral.topX;
427             newCorral.topY = oldCorral.topY;
428         }
429
430         return newCorral;
431     }
432
433     return corral1;
434 }
435
436 private int find(int i) {
437     if (parent[i] != i) {
438         int root = find(parent[i]);
439         parent[i] = root;

```

```

440         return root;
441     }
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
16     ↪ cost, int heuristic) {
17         super(playerPos, cratesIndices, hash, parent);
18         this.setCost(cost);
19         this.setHeuristic(heuristic);
20     }
21
22     public WeightedState(State state, int cost, int heuristic) {

```

```

22         this(state.playerPos(), state.cratesIndices(), state.hash(), state.parent(), cost,
23             ↪ heuristic);
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

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.poulpogaz.json.JsonException;

```

```

4 import fr.poulpogaz.json.JsonPrettyWriter;
5 import fr.valax.sokoshell.solver.board.Direction;
6 import fr.valax.sokoshell.solver.board.ImmutableBoard;
7 import fr.valax.sokoshell.solver.board.tiles.Tile;
8 import fr.valax.sokoshell.utils.BuilderException;
9 import fr.valax.sokoshell.utils.Utills;
10
11 import java.io.IOException;
12 import java.math.BigInteger;
13 import java.util.*;
14
15 /**
16  * @author darth-mole
17  * @author PoulpoGaz
18  */
19 public class Level extends ImmutableBoard {
20
21     // package private
22     Pack pack;
23     private final int playerPos;
24     private final int index;
25
26     private final List<SolverReport> solverReports;
27
28     // number of crate or crate on target
29     private final int numberOfCrates;
30
31     // number of crate, crate on target, floor and target
32     private final int numberOfNonWalls;
33
34     private BigInteger maxNumberOfStateEstimation;
35
36     public Level(Tile[][] tiles, int width, int height, int playerPos, int index) {
37         super(tiles, width, height);
38         this.playerPos = playerPos;
39         this.index = index;
40
41         solverReports = new ArrayList<>();
42
43         int numCrate = 0;
44         int numFloor = 0;
45         for (int y = 0; y < height; y++) {
46             for (int x = 0; x < width; x++) {
47                 if (getAt(x, y).anyCrate()) {
48                     numCrate++;
49                 }
50                 if (!getAt(x, y).isWall()) {
51                     numFloor++;
52                 }
53             }
54         }
55
56         this.numberOfCrates = numCrate;
57         this.numberOfNonWalls = numFloor;
58     }
59
60     public void writeSolutions(JsonPrettyWriter jpw) throws JSONException, IOException {
61         for (SolverReport solution : solverReports) {
62             jpw.beginObject();
63             solution.writeSolution(jpw);
64             jpw.endObject();
65         }
66     }

```

```

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 +
116                 ↵ 1);
117         }
118
119         return maxNumberOfStateEstimation;
120     }
121
122     public BigInteger estimateNumberOfState(int nDeadTile) {
123         int nFloor = numberOfNonWalls - nDeadTile;
124
125         return Utils.binomial(nFloor, numberOfCrates + 1);
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
215                 ↳ IllegalArgumentException("Attempting to add a report to the wrong level");
216         }
217         solverReports.add(solverReport);
218     }
219
220     public synchronized void removeSolverReport(int index) {
221         solverReports.remove(index);
222     }
223
224     public synchronized int indexOf(SolverReport solverReport) {
225         if (solverReport.getParameters().getLevel() != this) {
226             return -1;
227         }
228         return solverReports.indexOf(solverReport);
229     }
230
231     /**
232     * Returns if an attempt to solve this level was done. It doesn't mean that this level
233     ↳ has a solution
234     *
235     * @return {@code true} if an attempt to solve this level was done.
236     */
237     public boolean hasReport() {
238         return solverReports.size() > 0;
239     }
240
241     /**
242     * Returns {@code true} if this level has a solution
243     *
244     * @return {@code true} if this level has a solution
245     */
246     public boolean hasSolution() {
247         for (int i = 0; i < solverReports.size(); i++) {
248             SolverReport r = solverReports.get(i);
249             if (r.isSolved()) {
250                 return true;
251             }
252         }
253         return false;
254     }

```

```

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      * @throws BuilderException if 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     ↪ old one,
484         * Resize this level to (newWidth, newHeight). If dimensions are higher than the
485         * new tiles are filled with WALL. For other, tiles are the same.
486         *
487         * @param newWidth the new width of the level
488         * @param newHeight the new width of the level
489         */
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
21  * ↪ solver,
22  * some statistics, the solver status and if the status is {@link
23  * ↪ 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." +
109                 "You must give the solution");
110         }
111
112         SolutionBuilder builder = createFullSolution(states);
113
114         numberOfPushes = builder.getNumberOfPushes();
115         numberOfMoves = builder.getNumberOfMoves();
116         solution = builder.getSolution();
117     } else {
118         numberOfMoves = -1;
119         numberOfPushes = -1;
120         solution = null;
121     }
122 }
123
124 private SolverReport(SolverParameters parameters,
125                     ISolverStatistics statistics,
126                     String status,
127                     SolutionBuilder builder) {
128     this.parameters = Objects.requireNonNull(parameters);
129     this.statistics = Objects.requireNonNull(statistics);
130     this.status = Objects.requireNonNull(status);
131
132     if (status.equals(SOLUTION_FOUND)) {
133         numberOfPushes = builder.getNumberOfPushes();
134         numberOfMoves = builder.getNumberOfMoves();
135         solution = builder.getSolution();
136     } else {
137         numberOfMoves = -1;
138         numberOfPushes = -1;
139         solution = null;
140     }
141 }

```

```

142
143
144 /**
145  * Deduce from solution's states all the moves needed to solve the sokoban
146  *
147  * @return the full solution
148  */
149 private SolutionBuilder createFullSolution(List<State> states) {
150     Level level = parameters.getLevel();
151     Board board = new MutableBoard(level);
152
153     SolutionBuilder sb = new SolutionBuilder(2 * states.size());
154     List<Move> temp = new ArrayList<>();
155
156     TileInfo player = board.getAt(level.getPlayerX(), level.getPlayerY());
157
158     CrateAStar aStar = new CrateAStar(board);
159     for (int i = 0; i < states.size() - 1; i++) {
160         State current = states.get(i);
161
162         if (i != 0) {
163             board.addStateCrates(current);
164         }
165
166         State next = states.get(i + 1);
167         StateDiff diff = getStateDiff(board, current, next);
168
169         Node node = aStar.findPathAndComputeMoves(
170             player, null,
171             diff.crate(), diff.crateDest());
172
173         if (node == null) {
174             throw cannotFindPathException(board, current, next);
175         }
176
177         player = node.getPlayer();
178         while (node.getParent() != null) {
179             temp.add(node.getMove());
180             node = node.getParent();
181         }
182
183         sb.ensureCapacity(sb.getNumberOfMoves() + temp.size());
184         for (int j = temp.size() - 1; j >= 0; j--) {
185             sb.add(temp.get(j));
186         }
187         temp.clear();
188
189         board.removeStateCrates(current);
190     }
191
192     return sb;
193 }
194
195 /**
196  * Find the differences between two states:
197  * <ul>
198  *     <li>new player position</li>
199  *     <li>old crate pos</li>
200  *     <li>new crate pos</li>
201  * </ul>
202  *
203  * @param board the board

```

```

204 * @param from the first state
205 * @param to the second state
206 * @return a {@link StateDiff}
207 */
208 private StateDiff getStateDiff(Board board, State from, State to) {
209     List<Integer> state1Crates =
210         ↪ Arrays.stream(from.cratesIndices()).boxed().collect(Collectors.toList());
211     List<Integer> state2Crates =
212         ↪ Arrays.stream(to.cratesIndices()).boxed().collect(Collectors.toList());
213
214     List<Integer> state1Copy = state1Crates.stream().toList();
215     state1Crates.removeAll(state2Crates);
216     state2Crates.removeAll(state1Copy);
217
218     return new StateDiff(
219         board.getAt(to.playerPos()),
220         board.getAt(state1Crates.get(0)), // original crate pos
221         board.getAt(state2Crates.get(0)); // where it goes
222     }
223
224 /**
225  * Create an exception indicating a path can't be found between two states.
226  *
227  * @param board the board which must be in the same state as current
228  * @param current the current state
229  * @param next the next state
230  * @return an exception
231  */
232 private IllegalStateException cannotFindPathException(Board board, State current,
233     ↪ State next) {
234     BoardStyle style = SokoShell.INSTANCE.getBoardStyle();
235
236     String str1 = style.drawToString(board, board.getX(current.playerPos()),
237         ↪ board.getY(current.playerPos())).toAnsi();
238     board.removeStateCrates(current);
239     board.addStateCrates(next);
240     String str2 = style.drawToString(board, board.getX(next.playerPos()),
241         ↪ board.getY(next.playerPos())).toAnsi();
242
243     return new IllegalStateException("""
244         Can't find path between two states:
245         %s
246         (%s)
247         and
248         %s
249         (%s)
250         """).formatted(str1, current, str2, next);
251     }
252
253 public void writeSolution(JsonPrettyWriter jpw) throws JSONException, IOException {
254     jpw.field("status", status);
255     jpw.key("parameters");
256     parameters.append(jpw);
257
258     if (solution != null) {
259         jpw.key("solution").beginArray();
260         jpw.setInline(JsonPrettyWriter.Inline.ALL);
261
262         for (Move m : getFullSolution()) {

```

```

261         jpw.value(m.shortName());
262     }
263
264     jpw.endArray();
265     jpw.setInline(JsonPrettyWriter.Inline.NONE);
266 }
267
268 jpw.key("statistics");
269
270 // probably not a good way to do that, but I don't know
271 // how to easily serialize and deserialize ISolverStatistics
272 // without having a factory...
273 ByteArrayOutputStream baos = new ByteArrayOutputStream();
274 ObjectOutputStream oos = new ObjectOutputStream(baos);
275 oos.writeObject(statistics);
276 oos.close();
277
278 jpw.value(Base64.getEncoder().encodeToString(baos.toByteArray()));
279 }
280
281
282 public static SolverReport fromJson(JsonReader jr, Level level) throws JsonException,
↳ IOException {
283     String status = jr.assertKeyEquals("status").nextString();
284
285     jr.assertKeyEquals("parameters");
286     SolverParameters parameters = SolverParameters.fromJson(jr, level);
287
288     String key = jr.nextKey();
289
290     SolutionBuilder sb = null;
291     if (key.equals("solution")) {
292         jr.beginArray();
293
294         sb = new SolutionBuilder(32 * 5); // uses array of size 16
295         while (!jr.isArrayEnd()) {
296             String name = jr.nextString();
297             Move move = Move.of(name);
298
299             if (move == null) {
300                 throw new IOException("Unknown move: " + name);
301             }
302
303             sb.add(move);
304         }
305         jr.endArray();
306
307         jr.assertKeyEquals("statistics");
308     } else if (!key.equals("statistics")) {
309         throw new JsonException(String.format("Invalid key. " +
310             "Expected \"statistics\" but was \"%s\"", key));
311     }
312
313     // see writeSolution
314     byte[] bytes = Base64.getDecoder().decode(jr.nextString());
315
316     ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));
317     ISolverStatistics stats;
318     try {
319         stats = (ISolverStatistics) ois.readObject();
320     } catch (ClassNotFoundException e) {
321         throw new IOException(e);

```

```

322     }
323     ois.close();
324
325     return new SolverReport(parameters, stats, status, sb);
326 }
327
328
329 /**
330  * Returns the type of the solver used to produce this report
331  *
332  * @return the type of the solver used to produce this report
333  */
334 public String getSolverName() {
335     return parameters.getSolverName();
336 }
337
338 /**
339  * Returns the parameters given to the solver that produce this report
340  *
341  * @return the parameters given to the solver
342  */
343 public SolverParameters getParameters() {
344     return parameters;
345 }
346
347 /**
348  * Returns the statistics produce by the solver that produce this report.
349  * However, {@linkplain Solver solvers} are only capable of recording when
350  * the research start and end. Others statistics are produced by {@link Tracker}
351  *
352  * @return the parameters given to the solver
353  */
354 public ISolverStatistics getStatistics() {
355     return statistics;
356 }
357
358 public SolutionIterator getSolutionIterator() {
359     if (solution == null) {
360         return null;
361     }
362
363     return new SolutionIterator();
364 }
365
366 /**
367  * If the sokoban was solved, this report contains the solution as a sequence
368  * of moves. It describes all moves made by the player.
369  *
370  * @return the solution or {@code null} if the sokoban wasn't solved
371  */
372 public List<Move> getFullSolution() {
373     if (solution == null) {
374         return null;
375     }
376
377     ListIterator<Move> it = getSolutionIterator();
378     List<Move> moves = new ArrayList<>(numberOfMoves);
379
380     while (it.hasNext()) {
381         moves.add(it.next());
382     }
383

```

```

384         return moves;
385     }
386
387     /**
388      * Returns the number of pushes the player made to solve the sokoban
389      *
390      * @return {@code -1} if the sokoban wasn't solved or the number of pushes the player
↪ made to solve the sokoban
391      */
392     public int numberOfPushes() {
393         return numberOfPushes;
394     }
395
396     /**
397      * Returns the number of moves the player made to solve the sokoban
398      *
399      * @return {@code -1} if the sokoban wasn't solved or the number of moves the player
↪ made to solve the sokoban
400      */
401     public int numberOfMoves() {
402         return numberOfMoves;
403     }
404
405
406     /**
407      * Returns {@code true} if this report contains a solution
408      *
409      * @return {@code true} if this report contains a solution
410      */
411     public boolean isSolved() {
412         return status.equals(SOLUTION_FOUND);
413     }
414
415     /**
416      * Returns {@code true} if this report doesn't contain a solution
417      *
418      * @return {@code true} if this report doesn't contain a solution
419      */
420     public boolean hasNoSolution() {
421         return !status.equals(SOLUTION_FOUND);
422     }
423
424     /**
425      * Returns {@code true} if the solver was stopped by the user
426      *
427      * @return {@code true} if the solver was stopped by the user
428      */
429     public boolean isStopped() {
430         return status.equals(STOPPED);
431     }
432
433
434     public String getStatus() {
435         return status;
436     }
437
438     /**
439      * Returns the level that was given to the solver
440      *
441      * @return the level that was given to the solver
442      */
443     public Level getLevel() {

```

```

444         return parameters.getLevel();
445     }
446
447
448     /**
449     * Returns the pack of the level that was given to the solver
450     *
451     * @return the pack of the level that was given to the solver
452     */
453     public Pack getPack() {
454         return parameters.getLevel().getPack();
455     }
456
457     /**
458     * Contains all differences between two states except the old player position.
459     *
460     * @param playerDest player destination
461     * @param crate old crate position
462     * @param crateDest crate destination
463     */
464     private record StateDiff(TileInfo playerDest, TileInfo crate, TileInfo crateDest) {}
465
466     /**
467     * An object to iterate over a solution in forward and backward order.
468     */
469     public class SolutionIterator implements ListIterator<Move> {
470
471         /**
472         * Position in the array
473         */
474         private int arrayPos;
475
476         /**
477         * Position in solution[arrayPos]
478         */
479         private int bitPos;
480
481         private int move;
482         private int push;
483
484         /**
485         * @return read the next bit
486         */
487         private int readNext() {
488             int bit = (solution[arrayPos] >> bitPos) & 0b1;
489
490             bitPos++;
491             if (bitPos == 32) {
492                 bitPos = 0;
493                 arrayPos++;
494             }
495
496             return bit;
497         }
498
499         /**
500         * @return read the previous bit
501         */
502         private int readPrevious() {
503             bitPos--;
504             if (bitPos < 0) {
505                 bitPos = 31;

```

```

506         arrayPos--;
507     }
508
509     return (solution[arrayPos] >> bitPos) & 0b1;
510 }
511
512
513 @Override
514 public boolean hasNext() {
515     return move < numberOfMoves;
516 }
517
518 @Override
519 public Move next() {
520     if (!hasNext()) {
521         throw new NoSuchElementException();
522     }
523
524     int first = readNext();
525     int second = readNext();
526     int third = readNext();
527
528     int value = (third << 2) | (second << 1) | first;
529
530     Move move = Move.values()[value];
531
532     this.move++;
533     if (move.moveCrate()) {
534         push++;
535     }
536
537     return move;
538 }
539
540 @Override
541 public boolean hasPrevious() {
542     return move > 0;
543 }
544
545 @Override
546 public Move previous() {
547     if (!hasPrevious()) {
548         throw new NoSuchElementException();
549     }
550
551     int third = readPrevious();
552     int second = readPrevious();
553     int first = readPrevious();
554
555     int value = (third << 2) | (second << 1) | first;
556
557     Move move = Move.values()[value];
558
559     this.move--;
560     if (move.moveCrate()) {
561         push--;
562     }
563
564     return move;
565 }
566
567 @Override

```



```

568     public int nextIndex() {
569         return move;
570     }
571
572     @Override
573     public int previousIndex() {
574         return move - 1;
575     }
576
577     public void reset() {
578         move = 0;
579         arrayPos = 0;
580         bitPos = 0;
581     }
582
583     @Override
584     public void remove() {
585         throw new UnsupportedOperationException();
586     }
587
588     @Override
589     public void set(Move move) {
590         throw new UnsupportedOperationException();
591     }
592
593     @Override
594     public void add(Move move) {
595         throw new UnsupportedOperationException();
596     }
597
598     public int getMoveCount() {
599         return move;
600     }
601
602     public int getPushCount() {
603         return push;
604     }
605 }
606
607 /**
608  * A convenience object to convert a list of move to a solution array.
609  */
610 private static class SolutionBuilder {
611
612     private int[] solution;
613
614     private int arrayPos;
615     private int bitPos;
616
617     private int numberOfMoves;
618     private int numberOfPushes;
619
620     public SolutionBuilder(int estimatedNumberOfMove) {
621         solution = new int[computeArraySize(estimatedNumberOfMove)];
622     }
623
624     private void write(int bit) {
625         solution[arrayPos] = (bit & 0b1) << bitPos | solution[arrayPos];
626
627         bitPos++;
628         if (bitPos == 32) {
629             bitPos = 0;

```

```

630         arrayPos++;
631     }
632 }
633
634 public void add(Move move) {
635     if (bitPos + 3 >= 32 && arrayPos + 1 >= solution.length) {
636         ensureCapacity(numberOfMoves * 2 + 1);
637     }
638
639     int value = move.ordinal();
640     write(value & 0b1);
641     write((value >> 1) & 0b1);
642     write((value >> 2) & 0b1);
643     numberOfMoves++;
644
645     if (move.moveCrate()) {
646         numberOfPushes++;
647     }
648 }
649
650 public void ensureCapacity(int numberOfMove) {
651     int minArraySize = computeArraySize(numberOfMove);
652
653     if (minArraySize > solution.length) {
654         solution = Arrays.copyOf(solution, minArraySize);
655     }
656 }
657
658 public int getNumberOfMoves() {
659     return numberOfMoves;
660 }
661
662 public int getNumberOfPushes() {
663     return numberOfPushes;
664 }
665
666 public int[] getSolution() {
667     int arraySize = computeArraySize(numberOfMoves);
668
669     return Arrays.copyOf(solution, arraySize);
670 }
671
672 private int computeArraySize(int numberOfMove) {
673     int nBits = 3 * numberOfMove;
674
675     return nBits / 32 + 1;
676 }
677 }
678 }

```

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 FESSOSolver extends AbstractSolver<FESSOSolver.FESSOState> {
12
13     private Heuristic heuristic;
14     private int lowerBound;
15
16     public FESSOSolver() {
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() /
33             ↪ level.getWidth());
34         board.removeStateCrates(s);
35
36         lowerBound = heuristic.compute(s);
37
38         FESSOState state = new FESSOState(s, 0, lowerBound,
39             ↪ detector.getRealNumberOfCorral(), countPackedCrate(s));
40
41         toProcess.addState(state);
42     }
43
44     @Override
45     protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
46         if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
47             return;
48         }
49
50         final int i = board.topLeftReachablePosition(crate, crateDest);
51         // The new player position is the crate position
52         FESSOState s = toProcess.cachedState().child(i, crate.getCrateIndex(),
53             ↪ crateDest.getIndex());
54         s.setHeuristic(heuristic.compute(s));
55         s.setConnectivity(board.getCorralDetector().getRealNumberOfCorral());
56         s.setPacking(countPackedCrate(s));
57
58         if (processed.add(s)) {
59             toProcess.addState(s);
60         }
61     }
62
63     private int countPackedCrate(State state) {
64         int nPacked = 0;
65         for (int crate : state.cratesIndices()) {
66             TileInfo tile = board.getAt(crate);
67             if (tile.isTarget() || tile.isCrateOnTarget()) {
68                 nPacked++;
69             }
70         }
71
72         return nPacked;
73     }
74 }

```

```

70 }
71
72 @Override
73 public int lowerBound() {
74     return lowerBound;
75 }
76
77 private static class SolverPriorityQueue extends PriorityQueue<FESSOState>
78     implements SolverCollection<FESSOState> {
79
80     private FESSOState cachedState;
81
82     @Override
83     public void addState(FESSOState state) {
84         offer(state);
85     }
86
87     @Override
88     public FESSOState popState() {
89         return poll();
90     }
91
92     @Override
93     public FESSOState peekState() {
94         return peek();
95     }
96
97     @Override
98     public FESSOState peekAndCacheState() {
99         cachedState = popState();
100         return cachedState;
101     }
102
103     @Override
104     public FESSOState cachedState() {
105         return cachedState;
106     }
107 }
108
109 protected static class FESSOState extends WeightedState implements
110     ↪ Comparable<FESSOState> {
111
112     private int connectivity;
113     private int packing;
114
115     public FESSOState(int playerPos, int[] cratesIndices, int hash, State parent, int
116     ↪ cost, int heuristic) {
117         super(playerPos, cratesIndices, hash, parent, cost, heuristic);
118     }
119
120     public FESSOState(State state, int cost, int heuristic, int connectivity, int
121     ↪ packing) {
122         super(state, cost, heuristic);
123         this.connectivity = connectivity;
124         this.packing = packing;
125     }
126
127     @Override
128     public FESSOState child(int newPlayerPos, int crateToMove, int crateDestination) {
129         return new FESSOState(super.child(newPlayerPos, crateToMove,
130         ↪ crateDestination),
131             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(FESSOState 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(FESSOState a, FESSOState b) {
154         // -1 if this < o
155         // 0 if this = o
156         // 1 if this > o
157         if (a.packing > b.packing) {
158             return 1; // we want to maximize packing
159         } else if (a.packing < b.packing) {
160             return -1;
161         } else {
162             if (a.connectivity < b.connectivity) {
163                 return 1; // we want to minimize connectivity
164             } else if (a.connectivity > b.connectivity) {
165                 return -1;
166             } else {
167                 return Integer.compare(a.weight(), b.weight());
168             }
169         }
170     }
171 }
172 }

```

SolverParameters

```

1 package fr.valax.sokoshell.solver;
2
3 import fr.poulpogaz.json.IJsonReader;
4 import fr.poulpogaz.json.IJsonWriter;
5 import fr.poulpogaz.json.JsonException;
6 import fr.valax.sokoshell.SokoShell;
7
8 import java.io.IOException;
9 import java.util.*;
10
11 /**
12  * A collection of {@link SolverParameter} plus the name of the solver used and the level
13  * ↪ to
14  * solve. {@link Solver} known which level to solve thanks to this object

```

```

14  */
15  public class SolverParameters {
16
17      private final String solverName;
18      private final Level level;
19      private final Map<String, SolverParameter> parameters;
20
21      public SolverParameters(String solverName, Level level) {
22          this(solverName, level, null);
23      }
24
25      public SolverParameters(String solverName, Level level, List<SolverParameter>
26          ↪ parameters) {
27          this.solverName = Objects.requireNonNull(solverName);
28          this.level = Objects.requireNonNull(level);
29
30          if (parameters == null) {
31              this.parameters = Map.of();
32          } else {
33              this.parameters = new HashMap<>();
34
35              for (SolverParameter p : parameters) {
36                  this.parameters.put(p.getName(), p);
37              }
38          }
39
40      /**
41       * @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 getArguments(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
103     ↪ JSONException, IOException {
104         jr.beginObject();
105         String solverName = jr.assertKeyEquals("solver").nextString();
106
107         Solver solver = SokoShell.INSTANCE.getSolver(solverName);
108         if (solver == null) {
109             throw new IOException("No such solver: " + solverName);
110         }
111
112         List<SolverParameter> parameters = solver.getParameters();
113         while (!jr.isObjectEnd()) {
114             String key = jr.nextKey();
115
116             SolverParameter parameter = parameters.stream()
117                 .filter((s) -> s.getName().equals(key))
118                 .findFirst()
119                 .orElseThrow(() -> new IOException("No such parameter: " + key));
120
121             parameter.fromJson(jr);
122         }
123
124         jr.endObject();
125
126         return new SolverParameters(solverName, level, parameters);
127     }

```

FreezeDeadlockDetector

```

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.Tile;

```

```

6 import fr.valax.sokoshell.solver.board.tiles.TileInfo;
7
8 public class FreezeDeadlockDetector {
9
10     // http://www.sokobano.de/wiki/index.php?title=How_to_detect_deadlocks
11     public static boolean checkFreezeDeadlock(Board board, State state) {
12         int[] crates = state.cratesIndices();
13
14         for (int crate : crates) {
15             TileInfo info = board.getAt(crate);
16
17             if (checkFreezeDeadlock(info)) {
18                 return true;
19             }
20         }
21
22         return false;
23     }
24
25     public static boolean checkFreezeDeadlock(TileInfo crate) {
26         return crate.isCrate() &&
27             checkFreezeDeadlockRec(crate, Direction.LEFT) &&
28             checkFreezeDeadlockRec(crate, Direction.UP);
29     }
30
31     private static boolean checkFreezeDeadlockRec(TileInfo crate) {
32         return checkFreezeDeadlockRec(crate, Direction.LEFT) &&
33             checkFreezeDeadlockRec(crate, Direction.UP);
34     }
35
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.Utls;
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 }
```

```

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;
93         }
94
95         @Override
96         public int averageQueueSize() {
97             return -1;
98         }
99
100        @Override
101        public int lowerBound() {
102            return -1;
103        }
104    }
105 }

```

Pack

```

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
7 import java.io.*;
8 import java.nio.file.Files;
9 import java.nio.file.Path;
10 import java.nio.file.StandardOpenOption;
11 import java.util.Collections;
12 import java.util.List;
13 import java.util.Objects;
14 import java.util.zip.GZIPInputStream;

```

```

15 import java.util.zip.GZIPOutputStream;
16
17 /**
18  * A pack is a collection of levels with a name and an author
19  */
20 public final class Pack {
21
22     /**
23      * Some pack doesn't have a name while it is required by {@link
↵ fr.valax.sokoshell.SokoShell}.
24      * So pack without a name as named as following: 'Unnamed[I]' where I is an integer
↵ which is increased
25      * each time an unnamed pack is created. This 'I' is the variable below
26      */
27     private static int unnamedIndex = 0;
28
29
30     private final String name;
31     private final String author;
32     private final List<Level> levels;
33
34     private Path sourcePath;
35
36     public Pack(String name, String author, List<Level> levels) {
37         if (name == null) {
38             this.name = "Unnamed[" + unnamedIndex + "]";
39             unnamedIndex++;
40         } else {
41             this.name = name;
42         }
43
44         this.author = author;
45         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;
8 import java.util.Arrays;
9 import java.util.HashSet;
10 import java.util.Queue;
11
12 public class Hotspots {
13
14     private final Board board;
15
16     /**
17     * if hotspot[X][Y] is true then if there is a crate at Y and at X,
18     * Y blocks X to be pushed to at least one target
19     */
20     private final boolean[][] hotspot;
21
22     // Variables used by countAccessibleTargets
23     private ReachableTiles reachable;
24     // accessible[x] is true if at x there is target and the target is push-accessible
25     private boolean[] accessible;
26     private Queue<State> toVisit;
27     private HashSet<State> visited;
28
29     public Hotspots(Board board) {
30         this.board = board;
31         int s = board.getWidth() * board.getHeight();
32         this.hotspot = new boolean[s][s];

```

```

33 }
34
35 protected void postInit() {
36     int size = board.getWidth() * board.getHeight();
37     reachable = new ReachableTiles(board);
38     accessible = new boolean[size];
39     toVisit = new ArrayDeque<>();
40     visited = new HashSet<>();
41 }
42
43 public void computeHotspots() {
44     postInit();
45     int size = board.getWidth() * board.getHeight();
46
47     for (int X = 0; X < size; X++) {
48         TileInfo x = board.getAt(X);
49         if (x.isSolid()) {
50             continue;
51         }
52
53         x.addCrate();
54         int accessible = countAccessibleTargets(board, X);
55
56         for (int Y = 0; Y < size; Y++) {
57             if (Y == X) {
58                 continue;
59             }
60
61             TileInfo y = board.getAt(Y);
62             if (y.isSolid()) {
63                 continue;
64             }
65
66             y.addCrate();
67             int accessible2 = countAccessibleTargets(board, X);
68             y.removeCrate();
69
70             if (accessible != accessible2) {
71                 hotspot[X][Y] = true;
72             }
73         }
74         x.removeCrate();
75     }
76
77     reachable = null;
78     accessible = null;
79     toVisit = null;
80     visited = null;
81 }
82
83 protected int countAccessibleTargets(Board board, int baseCratePos) {
84     Arrays.fill(accessible, false);
85     toVisit.clear();
86     visited.clear();
87
88     // add base state
89     // There is four state, for each direction
90     TileInfo baseCrate = board.getAt(baseCratePos);
91     for (Direction d : Direction.VALUES) {
92         TileInfo player = baseCrate.adjacent(d);
93
94         if (!player.isSolid()) {

```

```

95         State s = new State(player, baseCrate);
96         toVisit.add(s);
97         visited.add(s);
98     }
99 }
100
101 if (baseCrate.isCrateOnTarget()) {
102     accessible[baseCratePos] = true;
103 }
104
105 baseCrate.removeCrate();
106
107 while (!toVisit.isEmpty()) {
108     State s = toVisit.poll();
109
110     s.crate().addCrate();
111     reachable.findReachableCases(s.player());
112
113     for (Direction dir : Direction.VALUES) {
114         TileInfo player = s.crate().adjacent(dir.negate());
115         TileInfo crateDest = s.crate().adjacent(dir);
116
117         if (crateDest.isSolid() || !reachable.isReachable(player)) {
118             continue;
119         }
120
121         if (crateDest.isTarget()) {
122             accessible[crateDest.getIndex()] = true;
123         }
124
125         int playerDest = board.topLeftReachablePosition(s.crate(), crateDest);
126
127         State newState = new State(board.getAt(playerDest), crateDest);
128         if (visited.add(newState)) {
129             toVisit.add(newState);
130         }
131     }
132
133     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) {}

```


