	idx.Sukusheti.Suivei	
pa	thfinder	
	PlayerAStar	2
	AStarMarkSystem	3
	AbstractAStar	5
	CratePlayerAStar	8
	Node	0
	CrateAStar	. 10
co	llections	
	Node	. 13
	SolverCollection	
	MinHeap	
	SolverPriorityQueue	. 17
he	uristic	4.0
	AbstractHeuristic	
	GreedyHeuristic	
	SimpleHeuristic	21
	Heuristic	.21
boa	ard	
	tiles	
	MutableTileInfo	. 22
	GenericTileInfo	
	ImmutableTileInfo	
	TileInfo	
	Tile	
		. 30
-	mark	
	HeavyweightMarkSystem	
	FixedSizeMarkSystem	
	Mark	. 38
	DefaultMark	. 38
	MarkSystem	. 39
	AbstractMarkSystem	.40
	Tunnel	
	ImmutableBoard	
	Move	
	Room	
	MutableBoard	
	Direction	
	GenericBoard	
	Board	
-	State	85
<u> </u>	ReachableTiles	. 89
	Solver	. 89
	DeadlockTable	. 90
	AStarSolver	. 98
	Corral	100
T	BruteforceSolver	
T	Tracker	
T	AbstractSolver	
T T		
ī	SolverParameter	
Ī	Trackable	
T	CorralDetector	
	WeightedState	
T	Level	
$\perp$	SolverReport	143
	FESSOSolver	154
T	SolverParameters	
T	FreezeDeadlockDetector	
T	ISolverStatistics	
T	Pack Pack	
<u> </u>		102

# 1 fr.valax.sokoshell.solver

# 1.1 pathfinder

## PlayerAStar

```
package fr.valax.sokoshell.solver.pathfinder;
   import fr.valax.sokoshell.solver.board.Board;
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.PriorityQueue;
9
   * An 'A*' that can find a path between a start position and an end position for a player.
   * It uses a local mark system.
11
12
   public class PlayerAStar extends AbstractAStar {
13
       private final int boardWidth;
15
       private final AStarMarkSystem markSystem;
16
       private final Node[] nodes;
17
18
       public PlayerAStar(Board board) {
19
           super(new PriorityQueue<>(board.getWidth() * board.getHeight()));
20
           this.boardWidth = board.getWidth();
21
           markSystem = new AStarMarkSystem(board.getWidth() * board.getHeight());
22
           nodes = new Node[board.getHeight() * board.getWidth()];
23
24
           for (int i = 0; i < nodes.length; i++) {</pre>
               nodes[i] = new Node();
           }
27
       }
28
       private int toIndex(TileInfo player) {
30
           return player.getY() * boardWidth + player.getX();
31
       }
32
       @Override
34
       protected void init() {
35
           markSystem.unmarkAll();
           queue.clear();
37
       }
38
39
       @Override
       protected void clean() {
42
43
44
       @Override
45
       protected Node initialNode() {
46
           int i = toIndex(playerStart);
47
           Node init = nodes[i];
49
           init.setInitial(playerStart, null, heuristic(playerStart));
50
           return init;
51
       }
53
       @Override
54
       protected Node processMove(Node parent, Direction dir) {
55
           TileInfo player = parent.getPlayer();
           TileInfo dest = player.adjacent(dir);
57
```

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

### AStarMarkSystem

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

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

```
public void markVisited() {
80
                mark = AStarMarkSystem.this.mark - 1;
81
82
            public void setVisited() {
                mark = AStarMarkSystem.this.mark;
85
86
            @Override
88
            public void unmark() {
89
                mark = AStarMarkSystem.this.mark - 2;
90
92
            @Override
93
            public boolean isMarked() {
                return mark == AStarMarkSystem.this.mark - 1;
96
97
            public boolean isVisited() {
98
                return mark == AStarMarkSystem.this.mark;
100
101
            @Override
            public MarkSystem getMarkSystem() {
103
                return AStarMarkSystem.this;
104
105
        }
   }
107
```

#### AbstractAStar

```
package fr.valax.sokoshell.solver.pathfinder;
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.Move;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.PriorityQueue;
9
    * Abstract implementation of A*.
10
11
   public abstract class AbstractAStar {
12
13
       protected TileInfo playerStart;
14
       protected TileInfo crateStart;
       protected TileInfo playerDest;
16
       protected TileInfo crateDest;
17
       protected final PriorityQueue<Node> queue;
19
20
       public AbstractAStar(PriorityQueue<Node> queue) {
21
           this.queue = queue;
22
       }
24
25
        * @return true if path exists
        * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
27
28
       public boolean hasPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart,
29
       → TileInfo crateDest) {
           return findPath(playerStart, playerDest, crateStart, crateDest) != null;
30
```

```
}
32
33
        * It also computes the move field in {@link Node}
35
        * @see #findPath(TileInfo, TileInfo, TileInfo, TileInfo)
36
        */
37
       public Node findPathAndComputeMoves(TileInfo playerStart, TileInfo playerDest,
           TileInfo crateStart, TileInfo crateDest) {
           Node end = findPath(playerStart, playerDest, crateStart, crateDest);
39
40
           if (end == null) {
               return null;
42
43
           Node current = end;
           while (current.getParent() != null) {
46
               Node last = current.getParent();
47
               TileInfo lastPlayer = last.getPlayer();
               TileInfo currPlayer = current.getPlayer();
50
               Direction dir = Direction.of(currPlayer.getX() - lastPlayer.getX(),
51

    currPlayer.getY() - lastPlayer.getY());
52
               boolean moved = crateStart != null &&
53
                !current.getCrate().isAt(last.getCrate());
               current.setMove(Move.of(dir, moved));
55
               current = last;
56
           }
57
           return end;
59
       }
60
61
        * Find a path between (playerStart, crateStart) and (playerDest, crateDest).
        * The returned node may be cached by the implementation. Therefore, if you
64
        * want to keep the path in memory, you need to copy the path.
65
        * Oparam playerStart player start
67
        * @param playerDest player dest
68
        * Oparam crateStart crate start
        * Oparam crateDest crate dest
        * Oreturn the shortest path as a linked list in reverse.
71
72
       public Node findPath(TileInfo playerStart, TileInfo playerDest, TileInfo crateStart,
       → TileInfo crateDest) {
           this.playerStart = playerStart;
74
           this.crateStart = crateStart;
75
           this.playerDest = playerDest;
           this.crateDest = crateDest;
77
78
           init();
           Node n = initialNode();
           queue.offer(n);
81
82
           // int c = 0;
83
           Node end = null;
           while (!queue.isEmpty()) {
               Node node = queue.poll();
86
87
               if (isEndNode(node)) {
```

```
end = node;
89
                     break;
90
                }
91
92
                if (isVisited(node)) {
93
                     continue;
94
                }
95
                for (Direction direction : Direction.VALUES) {
97
                     Node child = processMove(node, direction);
98
99
                     if (child != null) {
                         queue.offer(child);
101
102
                }
103
                markVisited(node);
105
                // c++;
106
107
            // System.out.println(c);
109
            clean();
110
            return end;
111
        }
113
114
         * Decrease the priority of the node in the queue if and only if it is in the queue
         * Oparam node node
116
         */
117
        public void decreasePriority(Node node) {
118
            // TODO: we do not have a fixed size binary heap that
            // can efficiently decrease priority (at least O(log n))
120
            if (queue.remove(node)) { // takes O(n)
121
                queue.offer(node); // takes O(log n)
122
            }
123
        }
124
125
126
         * Init A*. Usually clear the queue. Called before the search
128
        protected abstract void init();
129
        /**
         * Clean the object. Called at the end of the search
132
133
        protected abstract void clean();
134
135
        /**
136
         * Returns the initial node.
137
         * Oreturn the initial node
         */
139
        protected abstract Node initialNode();
140
141
        /**
142
         * Oparam parent parent node
144
         * Oparam dir direction taken player
145
         * @return {@code null} if the player cannot move in the specified direction
         * or if the node was already visited. Otherwise, returns child node
147
148
        protected abstract Node processMove(Node parent, Direction dir);
149
150
```

```
/**
         * Mark the node as visited
152
         * @param node node
153
154
        protected abstract void markVisited(Node node);
155
156
157
         * @param node node
         * @return {@code true} if the node is visited
159
160
        protected abstract boolean isVisited(Node node);
161
        /**
163
         * Oparam node node
164
         * @return {@code true} if this node represents the solution
165
         */
166
        protected abstract boolean isEndNode(Node node);
167
168
```

### CratePlayerAStar

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

```
40 | }
41 |}
```

#### Node

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

```
this.dist = newParent.dist + 1;
58
59
            expectedDist = dist + heuristic;
60
        }
61
62
        public Node getParent() {
63
            return parent;
64
66
        public int getDist() {
67
            return dist;
68
        }
70
        public int getHeuristic() {
71
            return heuristic;
72
        }
73
74
        public TileInfo getPlayer() {
75
            return player;
76
77
        }
78
        public TileInfo getCrate() {
79
            return crate;
        }
81
82
        public Move getMove() {
83
            return move;
        }
85
86
        public void setMove(Move move) {
87
            this.move = move;
89
90
        public int getExpectedDist() {
91
            return expectedDist;
        }
93
94
        @Override
95
        public boolean equals(Object o) {
            if (this == o) return true;
97
            if (!(o instanceof Node node)) return false;
98
99
            if (!Objects.equals(player, node.player)) return false;
100
            return Objects.equals(crate, node.crate);
101
        }
102
103
        @Override
104
        public int hashCode() {
105
            int result = player != null ? player.getIndex() : 0;
106
            result = 31 * result + (crate != null ? crate.getIndex() : 0); // TODO
107
            return result;
108
        }
109
110
        @Override
111
        public int compareTo(Node o) {
            return Integer.compare(expectedDist, o.expectedDist);
113
        }
114
   }
```

### CrateAStar

```
package fr.valax.sokoshell.solver.pathfinder;
2
   import fr.valax.sokoshell.solver.board.Board;
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.PriorityQueue;
    * Moves a crate from a start position to a destination.
10
11
   public class CrateAStar extends AbstractAStar {
13
       private final int boardWidth;
14
       private final int area;
15
16
       private final AStarMarkSystem markSystem;
17
       private final Node[] nodes;
18
19
       public CrateAStar(Board board) {
           super(new PriorityQueue<>(2 * board.getWidth() * board.getHeight()));
21
           this.boardWidth = board.getWidth();
22
           area = board.getWidth() * board.getHeight();
           markSystem = new AStarMarkSystem(area * area);
25
26
           nodes = new Node[area * area];
           for (int i = 0; i < nodes.length; i++) {</pre>
29
               nodes[i] = new Node();
30
           }
       }
32
33
       private int toIndex(TileInfo player, TileInfo crate) {
34
           return (player.getY() * boardWidth + player.getX()) * area + crate.getY() *
              boardWidth + crate.getX();
       }
36
       @Override
       protected void init() {
39
           markSystem.unmarkAll();
40
           queue.clear();
41
           crateStart.removeCrate();
       }
43
44
       @Override
       protected void clean() {
46
           crateStart.addCrate();
47
48
49
       @Override
50
       protected Node initialNode() {
51
           int i = toIndex(playerStart, crateStart);
52
           Node init = nodes[i];
54
           init.setInitial(playerStart, crateStart, heuristic(playerStart, crateStart));
55
           return init;
56
       }
57
58
       @Override
59
       protected Node processMove(Node parent, Direction dir) {
```

```
TileInfo player = parent.getPlayer();
61
            TileInfo crate = parent.getCrate();
62
            TileInfo playerDest = player.adjacent(dir);
63
            TileInfo crateDest = crate;
65
            if (playerDest.isAt(crate)) {
66
                crateDest = playerDest.adjacent(dir);
67
                if (crateDest.isSolid()) {
69
                    return null;
70
71
                // check deadlock
73
                if (!crateDest.isAt(this.crateDest) && // not a deadlock is if is destination
                         crateDest.adjacent(dir).isSolid() && // front must be solid
                         (crateDest.adjacent(dir.left()).isSolid() || // perp must be solid
                                 crateDest.adjacent(dir.right()).isSolid())) {
77
                    return null;
78
                }
79
            } else if (playerDest.isSolid()) {
                return null;
81
82
            int i = toIndex(playerDest, crateDest);
            Node node = nodes[i];
85
86
            if (markSystem.isMarked(i) || markSystem.isVisited(i)) {
                if (parent.getDist() + 1 + node.getHeuristic() < node.getExpectedDist()) {</pre>
                    node.changeParent(parent);
89
                    decreasePriority(node);
90
                }
92
                return null;
93
            } else {
94
                markSystem.mark(i);
95
                node.set(parent, playerDest, crateDest, heuristic(playerDest, crateDest));
97
                return node;
            }
       }
100
101
       @Override
102
       protected void markVisited(Node node) {
103
            markSystem.setVisited(toIndex(node.getPlayer(), node.getCrate()));
104
105
106
        @Override
107
       protected boolean isVisited(Node node) {
108
            return markSystem.isVisited(toIndex(node.getPlayer(), node.getCrate()));
109
        }
110
111
        @Override
112
       protected boolean isEndNode(Node node) {
113
            return node.getCrate().isAt(crateDest);
       }
116
       protected int heuristic(TileInfo newPlayer, TileInfo newCrate) {
117
            int h = newCrate.manhattanDistance(crateDest);
119
            /* the player first need to move near the crate to push it
120
               may not be optimal for level like this:
121
122
```

```
########
124
                # ##### #
125
                 # ##### #
126
                 # ##### #
127
                 0$ # The player needs to do a detour to push the crate
128
                 # #######
129
            if (newPlayer.manhattanDistance(newCrate) > 1) {
131
                h += newPlayer.manhattanDistance(newCrate);
132
            }
133
            return h;
135
        }
136
   }
137
```

### 1.2 collections

#### Node

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

```
this.value = value;

this.value = value;

}
```

#### SolverCollection

```
package fr.valax.sokoshell.solver.collections;
   import fr.valax.sokoshell.solver.State;
   public interface SolverCollection<T extends State> {
       void clear();
       boolean isEmpty();
10
       int size();
12
       void addState(T state);
13
       T popState();
16
       T peekState();
17
       T peekAndCacheState();
19
20
       T cachedState();
21
22
   }
```

# MinHeap

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

```
* Oparam capacity The (fixed) capacity of the heap
29
30
       public MinHeap(int capacity) {
31
           nodes = new ArrayList<>(capacity);
           for (int i = 0; i < capacity; i++) {
33
                nodes.add(i, new Node<T>());
34
35
           currentSize = 0;
       }
37
38
       protected int leftChild(int i) {
39
           return 2 * i + 1;
41
42
       protected int rightChild(int i) {
43
           return 2 * i + 2;
45
46
       protected void moveNodeUp(int i) {
47
           if (i == 0) {
                return;
49
           }
50
           final int p = parent(i);
           if (nodes.get(i).hasPriorityOver(nodes.get(p))) {
                Collections.swap(nodes, i, p);
53
                moveNodeUp(p);
54
           }
       }
56
57
       protected void moveNodeDown(int i) {
           int j = i;
           final int l = leftChild(i), r = rightChild(i);
60
           if (1 < size() && nodes.get(1).hasPriorityOver(nodes.get(i))) {</pre>
61
                j = 1;
62
           }
           if (r < size() && nodes.get(r).hasPriorityOver(nodes.get(l))) {</pre>
                j = r;
65
           }
           if (i != j) {
68
                Collections.swap(nodes, i, j);
69
               moveNodeDown(j);
70
           }
       }
72
73
       private int parent(int i) {
74
           assert i != 0;
           return (i - 1) / 2;
76
77
       public void add(T content, int priority) {
79
           int i = 0;
80
           if (currentSize == -1) {
                nodes.add(new Node<>(content, priority));
                moveNodeUp(nodes.size() - 1);
83
84
                nodes.get(currentSize).set(content, priority);
85
                moveNodeUp(currentSize);
                currentSize++;
87
           }
88
       }
89
```

```
public T pop() {
91
            final int i = size() - 1;
92
            Collections.swap(nodes, 0, i);
93
            T content;
            if (currentSize == -1) {
95
                 content = nodes.remove(i).content();
96
            } else {
97
                 content = nodes.get(i).content();
                 currentSize--;
99
            }
100
            moveNodeDown(0);
101
            return content;
        }
103
104
        public T peek() {
105
            return nodes.get(0).content();
107
108
        public void clear() {
109
            if (currentSize == - 1) {
                 nodes.clear();
111
            } else {
112
                 currentSize = 0;
113
            }
        }
115
116
        public boolean isEmpty() {
117
            return currentSize == -1 ? nodes.isEmpty() : (currentSize == 0);
118
119
120
        public int size() {
121
            return currentSize == -1 ? nodes.size() : currentSize;
122
123
124
        /**
125
         * Min heap (state, priority) couple.
126
127
        protected static final class Node<T> {
128
            private T content;
            private int priority;
130
131
            public Node() {
132
                 set(null, Integer.MAX_VALUE);
            }
134
135
            public Node(T content, int priority) {
136
                 set(content, priority);
137
138
139
            public boolean hasPriorityOver(Node<T> o) {
                 return priority < o.priority;</pre>
141
142
143
            @Override
144
            public String toString() {
                 return String.format("Node[priority=%d]", priority);
146
            }
147
            public void set(T content, int priority) {
149
                 this.content = content;
150
                 this.priority = priority;
151
            }
```

```
public T content() {
154
                 return content;
155
             }
157
             public void setContent(T content) {
158
                 this.content = content;
159
161
             public int priority() {
162
                 return priority;
163
             }
165
             public void setPriority(int priority) {
166
                 this.priority = priority;
167
             }
        }
169
170
```

### SolverPriorityQueue

```
package fr.valax.sokoshell.solver.collections;
   import fr.valax.sokoshell.solver.WeightedState;
3
4
   * Priority queue of dynamic capacity. The priority are in <strong>ASCENDANT</strong>
   → order, i.e. the element returned
    * by {@link SolverPriorityQueue#popState()} with the <strong>LOWEST</strong> priority.
   public class SolverPriorityQueue implements SolverCollection<WeightedState> {
10
11
        * @implNote We use a min heap collection.
13
       private final MinHeap<WeightedState> heap = new MinHeap<>();
14
15
       private WeightedState cachedState;
17
       @Override
18
       public void addState(WeightedState state) {
19
           heap.add(state, state.weight());
21
22
       @Override
23
       public WeightedState popState() {
           return heap.pop();
25
26
27
       @Override
       public WeightedState peekState() {
29
           return heap.peek();
30
       }
31
32
       @Override
33
       public WeightedState peekAndCacheState() {
34
           cachedState = popState();
           return cachedState;
36
       }
37
38
       @Override
       public WeightedState cachedState() {
40
```

```
return cachedState;
41
       }
42
43
       @Override
44
       public void clear() {
45
            heap.clear();
46
47
       @Override
49
       public boolean isEmpty() {
50
            return heap.isEmpty();
51
53
       @Override
54
       public int size() {
            return heap.size();
57
   }
58
```

### 1.3 heuristic

### AbstractHeuristic

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

# GreedyHeuristic

```
package fr.valax.sokoshell.solver.heuristic;
  import fr.valax.sokoshell.solver.State;
  import fr.valax.sokoshell.solver.board.Board;
  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
6
   * According to <a
   → href="http://sokobano.de/wiki/index.php?title=Solver#Greedy_approach">this article</a>
  public class GreedyHeuristic extends AbstractHeuristic {
10
11
      private final LinkedList list;
12
13
      public GreedyHeuristic(Board board) {
14
           super(board);
           final int n = board.getTargetCount();
16
```

```
17
           list = new LinkedList(n);
18
       }
19
20
       @Override
21
       public int compute(State s) {
22
           int heuristic = 0;
23
           board.getMarkSystem().unmarkAll();
25
26
           int n = 0;
27
           for (int crate : s.cratesIndices()) {
                TileInfo tile = board.getAt(crate);
29
30
                if (tile.isCrateOnTarget()) {
31
                    tile.mark();
                } else {
33
                    list.add(tile);
34
35
                    n++;
                }
37
           }
38
40
           for (int i = 0; i < n; i++) {
41
                Node minNode = list.getHead();
42
                TileInfo.TargetRemoteness minDist = minNode.getNearestNotAttributedTarget();
43
44
                Node node = minNode.nextNode();
45
                while (node != null) {
46
                    TileInfo.TargetRemoteness nearest = node.getNearestNotAttributedTarget();
47
48
                    if (nearest.distance() < minDist.distance()) {</pre>
49
                        minNode = node;
50
                        minDist = nearest;
51
                    }
53
                    node = node.nextNode();
54
                }
56
                board.getAt(minDist.index()).mark();
57
                minNode.getCrate().mark();
                heuristic += minDist.distance();
60
                minNode.remove();
61
           }
62
           return heuristic;
64
       }
65
       private static class LinkedList {
67
68
           private final Node[] nodeCache;
69
           private int size = 0;
70
           private Node head;
72
73
           public LinkedList(int size) {
                nodeCache = new Node[size];
76
                for (int i = 0; i < size; i++) {
77
                    nodeCache[i] = new Node(this);
```

```
}
79
            }
80
81
             public void add(TileInfo crate) {
                 Node newHead = nodeCache[size];
83
                 newHead.set(crate);
84
85
                 if (head != null) {
                     newHead.next = head;
87
                     head.previous = newHead;
88
89
                 head = newHead;
91
                 size++;
92
            }
93
             public void remove(Node node) {
95
                 if (node == head) {
96
                     head = node.next;
97
                     if (head != null) {
99
                          head.previous = null;
100
                     }
101
                 } else {
102
                     node.previous.next = node.next;
103
104
                     if (node.next != null) {
105
                          node.next.previous = node.previous;
106
107
                 }
108
                 size--;
110
            }
111
112
             public Node getHead() {
113
                 return head;
114
115
        }
116
        private static class Node {
118
119
            private final LinkedList list;
120
            private TileInfo crate;
122
            private Node previous;
123
            private Node next;
124
125
126
              * Index in crate's target remoteness
127
             private int index = 0;
129
130
            public Node(LinkedList list) {
131
                 this.list = list;
132
134
             public void set(TileInfo tile) {
135
                 crate = tile;
                 index = 0;
137
138
139
            public void remove() {
```

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

## **SimpleHeuristic**

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

### Heuristic

```
package fr.valax.sokoshell.solver.heuristic;
import fr.valax.sokoshell.solver.State;
/**
```

```
* Heuristic computing class for guided-search (e.g. A*)

*/
public interface Heuristic {

/**

* Computes the heuristic of the given state.

* Oparam s the state to compute the heuristic

* Oreturn the heuristic of the state

*/
int compute(State s);

}
```

# 1.4 board

#### 1.4.1 tiles

### MutableTileInfo

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

```
* The index of this crate in the {@link State#cratesIndices()} array
45
46
       protected int crateIndex;
47
48
49
        // Dynamic information
50
        protected Mark reachable;
       protected Mark mark;
52
53
       public MutableTileInfo(MutableBoard board, Tile tile, int x, int y) {
54
            super(board, tile, x, y);
            this.board = board;
56
57
            this.reachable = board.getReachableMarkSystem().newMark();
            this.mark = board.getMarkSystem().newMark();
       }
60
61
       public MutableTileInfo(MutableBoard board, TileInfo other) {
62
            super(board, other);
            this.board = board;
64
65
            this.reachable = board.getReachableMarkSystem().newMark();
            this.mark = board.getMarkSystem().newMark();
67
       }
68
69
       // GETTERS //
71
        @Override
72
       public boolean isDeadTile() {
73
            return deadTile;
75
76
        @Override
77
       public boolean isReachable() {
            return !tile.isSolid() && board.getCorral(this).containsPlayer();
80
81
        @Override
       public Tunnel getTunnel() {
83
            return tunnel;
84
       }
85
        @Override
87
       public Tunnel.Exit getTunnelExit() {
88
            return tunnelExit;
89
91
       public boolean isInATunnel() {
92
            return tunnel != null;
       }
94
95
        @Override
96
       public Room getRoom() {
97
            return room;
98
99
100
       @Override
       public boolean isInARoom() {
102
            return room != null;
103
       }
104
105
```

```
@Override
        public boolean isMarked() {
107
            return mark.isMarked();
108
110
        @Override
111
        public TargetRemoteness getNearestTarget() {
112
            return nearestTarget;
114
115
        @Override
116
        public TargetRemoteness[] getTargets() {
            return targets;
118
119
120
        // SETTERS //
122
123
        @Override
124
        public void addCrate() {
            if (tile == Tile.FLOOR) {
126
                tile = Tile.CRATE;
127
            } else if (tile == Tile.TARGET) {
                tile = Tile.CRATE_ON_TARGET;
130
        }
131
        @Override
133
        public void removeCrate() {
134
            if (tile == Tile.CRATE) {
135
                 tile = Tile.FLOOR;
            } else if (tile == Tile.CRATE_ON_TARGET) {
137
                 tile = Tile.TARGET;
138
139
        }
140
        @Override
142
        public void setTile(Tile tile) {
143
            this.tile = tile;
145
146
        @Override
147
        public void setDeadTile(boolean deadTile) {
            this.deadTile = deadTile;
149
150
151
        @Override
        public void setReachable(boolean reachable) {
153
            this.reachable.setMarked(reachable);
154
        }
155
156
        @Override
157
        public void setTunnel(Tunnel tunnel) {
158
            this.tunnel = tunnel;
159
161
        @Override
162
        public void setTunnelExit(Tunnel.Exit tunnelExit) {
            this.tunnelExit = tunnelExit;
164
165
166
        @Override
```

```
public void setRoom(Room room) {
            this.room = room;
169
170
171
        @Override
172
        public void mark() {
173
            mark.mark();
174
176
        @Override
177
        public void unmark() {
178
            mark.unmark();
180
181
        @Override
182
        public void setMarked(boolean marked) {
            mark.setMarked(marked);
184
185
186
        @Override
        public void setTargets(TargetRemoteness[] targets) {
188
            this.targets = targets;
189
        @Override
192
        public void setNearestTarget(TargetRemoteness nearestTarget) {
193
            this.nearestTarget = nearestTarget;
194
195
196
        @Override
197
        public int getCrateIndex() {
            return crateIndex;
199
200
201
        @Override
202
        public void setCrateIndex(int crateIndex) {
            this.crateIndex = crateIndex;
204
205
   }
```

### GenericTileInfo

```
package fr.valax.sokoshell.solver.board.tiles;
2
   import fr.valax.sokoshell.solver.board.Board;
   import fr.valax.sokoshell.solver.board.Room;
   import fr.valax.sokoshell.solver.board.Tunnel;
   * A {@code package-private} class meant to be use as a base class for {@link TileInfo}
   \hookrightarrow implementations.
    * It defines all the basic properties and their corresponding getters
   * (position, tile, board, etc.)
    * @see TileInfo
12
13
   public abstract class GenericTileInfo implements TileInfo {
15
       protected final Board board;
16
17
       protected final int x;
       protected final int y;
19
```

```
20
       protected Tile tile;
21
22
       /**
        * Create a new TileInfo
25
26
        * Oparam tile the tile
        * Oparam x the position on the x-axis in the board
28
        * Oparam y the position on the y-axis in the board
29
30
       public GenericTileInfo(Board board, Tile tile, int x, int y) {
           this.board = board;
32
           this.tile = tile;
33
           this.x = x;
           this.y = y;
       }
36
37
       public GenericTileInfo(TileInfo tileInfo) {
38
           this(tileInfo.getBoard(), tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
40
41
       public GenericTileInfo(Board board, TileInfo tileInfo) {
           this(board, tileInfo.getTile(), tileInfo.getX(), tileInfo.getY());
43
44
45
       @Override
       public Tile getTile() {
47
           return tile;
48
49
       @Override
51
       public int getX() {
52
           return x;
53
       }
54
       @Override
56
       public int getY() {
57
           return y;
59
60
61
        * Returns the board in which this tile is
63
        * Oreturn the board in which this tile is
64
        */
       public Board getBoard() {
           return board;
67
68
       // SETTERS: throw UnsupportedOperationException as this class is immutable //
70
71
       @Override
72
       public void addCrate() {
           throw new UnsupportedOperationException("Immutable object");
75
76
       @Override
       public void removeCrate() {
           throw new UnsupportedOperationException("Immutable object");
79
       }
80
```

```
@Override
        public void setTile(Tile tile) {
83
            throw new UnsupportedOperationException("Immutable object");
84
        }
86
        @Override
87
       public void setDeadTile(boolean deadTile) {
            throw new UnsupportedOperationException("Immutable object");
90
91
        @Override
92
        public void setReachable(boolean reachable) {
            throw new UnsupportedOperationException("Immutable object");
94
95
        Onverride
       public void setTunnel(Tunnel tunnel) {
98
            throw new UnsupportedOperationException("Immutable object");
99
       }
100
        @Override
102
       public void setTunnelExit(Tunnel.Exit tunnelExit) {
103
            throw new UnsupportedOperationException("Immutable object");
105
106
        @Override
107
       public void setRoom(Room room) {
            throw new UnsupportedOperationException("Immutable object");
109
110
111
        @Override
       public void mark() {
113
            throw new UnsupportedOperationException("Immutable object");
114
       }
115
116
        @Override
117
       public void unmark() {
118
            throw new UnsupportedOperationException("Immutable object");
119
121
        @Override
122
       public void setMarked(boolean marked) {
            throw new UnsupportedOperationException("Immutable object");
        }
125
126
        @Override
127
       public void setTargets(TargetRemoteness[] targets) {
128
            throw new UnsupportedOperationException("Immutable object");
129
        }
130
        @Override
132
       public void setNearestTarget(TargetRemoteness nearestTarget) {
133
            throw new UnsupportedOperationException("Immutable object");
134
        @Override
137
       public void setCrateIndex(int index) {
138
            throw new UnsupportedOperationException("Immutable object");
        }
140
141
        @Override
142
       public int hashCode() {
```

#### ImmutableTileInfo

```
package fr.valax.sokoshell.solver.board.tiles;
   import fr.valax.sokoshell.solver.board.ImmutableBoard;
3
   import fr.valax.sokoshell.solver.board.Room;
   import fr.valax.sokoshell.solver.board.Tunnel;
   * Immutable implementation of {@link TileInfo}.
   * This class basically extends {@link GenericTileInfo}. It implements the setters methods
10
   \hookrightarrow defined in
   * {@link TileInfo} by throwing an {@link UnsupportedOperationException}.
11
   * It also implements the 'solver-intended' properties by always returning the default
   → value: for instance, a
   * {@link ImmutableTileInfo} is never a 'dead tile', so the {@link #isDeadTile} method
   → will always return {@code false}.
   * The same policy is applied for each property.
15
    * @see TileInfo
16
   * @see GenericTileInfo
17
   public class ImmutableTileInfo extends GenericTileInfo {
19
20
       public ImmutableTileInfo(ImmutableBoard board, Tile tile, int x, int y) {
21
           super(board, tile, x, y);
23
24
       public ImmutableTileInfo(TileInfo tileInfo) {
           super(tileInfo);
26
27
28
       // GETTERS //
30
       @Override
31
       public boolean isDeadTile() {
32
           return false;
34
35
       @Override
       public boolean isReachable() {
           return true;
38
39
40
       @Override
       public Tunnel getTunnel() {
42
           return null;
43
       }
44
       @Override
46
       public Tunnel.Exit getTunnelExit() {
47
           return null;
49
50
       @Override
51
       public boolean isInATunnel() {
           return false;
53
```

```
}
55
       @Override
56
       public Room getRoom() {
57
            return null;
59
60
       @Override
       public boolean isInARoom() {
62
            return false;
63
64
       @Override
66
       public boolean isMarked() {
67
            return false;
70
       @Override
71
       public String toString() {
72
            return tile.toString();
74
75
       @Override
       public TargetRemoteness getNearestTarget() {
77
            return null;
78
79
       @Override
81
       public TargetRemoteness[] getTargets() {
82
            return null;
83
       }
85
       @Override
86
       public int getCrateIndex() {
87
           return -1;
       }
89
   }
90
```

# TileInfo

```
package fr.valax.sokoshell.solver.board.tiles;
  import fr.valax.sokoshell.solver.Corral;
3
  import fr.valax.sokoshell.solver.board.*;
  import fr.valax.sokoshell.solver.board.mark.Mark;
  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
  import java.util.List;
10
   * The {@link TileInfo} interface defines the methods that {@link Board} implementations
   → need to manage tiles,
   * for instance:
   * 
         the position
14
         the {@link Tile}
15
   * 
   * It defines a set of high-level interactions functions.
17
18
   * @see Board
19
   */
21 public interface TileInfo {
```

```
// GETTERS //
23
24
       * @return the position of this TileInfo on the x-axis
27
       int getX();
28
       /**
30
       * Oreturn the position of this TileInfo on the y-axis
31
32
       int getY();
34
35
       * @return which tile is this TileInfo
36
        */
       Tile getTile();
38
39
       /**
40
        * Oreturn true if there is a crate at this position
42
       default boolean anyCrate() {
43
           return getTile().isCrate();
44
46
47
       * @return true if there is a wall or a crate at this position
       default boolean isSolid() {
50
           return getTile().isSolid();
51
       }
53
54
        * Oreturn true if this TileInfo is exactly a floor
55
       */
       default boolean isFloor() {
57
          return getTile() == Tile.FLOOR;
58
       }
59
61
       * Oreturn true if this TileInfo is exactly a wall
62
       default boolean isWall() {
           return getTile() == Tile.WALL;
65
66
67
       /**
       * Oreturn true if this TileInfo is exactly a target
69
70
       default boolean isTarget() {
           return getTile() == Tile.TARGET;
72
       }
73
74
75
       * @return true if this TileInfo is exactly a crate
       * @see #anyCrate()
77
78
       default boolean isCrate() {
           return getTile() == Tile.CRATE;
80
81
82
       /**
83
```

```
* Oreturn true if this TileInfo is exactly a crate on target
         * @see #anyCrate()
85
        */
86
        default boolean isCrateOnTarget() {
87
            return getTile() == Tile.CRATE_ON_TARGET;
89
90
        * Returns {@code true} if this tile is at the same position as 'other'
92
         * Oparam other other tile
93
        * Oreturn {Ocode true} if this tile is at the same position as 'other'
94
        default boolean isAt(TileInfo other) {
96
            return isAt(other.getX(), other.getY());
97
        }
98
        /**
100
         * Returns {@code true} if this tile is at the position (x; y)
101
         * @param x x location
102
        * Oparam y y location
         * Oreturn {Ocode true} if this tile is at the position (x; y)}
104
        */
105
        default boolean isAt(int x, int y) {
106
107
            return x == getX() && y == getY();
        }
108
109
        * Returns the direction between this tile and other.
111
112
        * Oparam other 'other' tile
113
        * Oreturn the direction between this tile and other
114
115
        default Direction direction(TileInfo other) {
116
            return Direction.of(other.getX() - getX(), other.getY() - getY());
117
        }
118
119
        /**
120
        * Returns the distance of manhattan between this tile and other
121
        * Oparam other 'other' tile
123
        * @return the distance of manhattan between this tile and other
124
        default int manhattanDistance(TileInfo other) {
            return Math.abs(getX() - other.getX()) + Math.abs(getY() - other.getY());
127
128
129
        * Oreturn {Ocode true} if this tile is a dead tile
131
         * @see MutableBoard#computeDeadTiles()
132
        */
133
        boolean isDeadTile();
134
135
136
        * Oreturn {Ocode true} if this tile is reachable by the player.
137
        * @see MutableBoard#findReachableCases(int)
139
        boolean isReachable();
140
        /**
142
        * Returns the tunnel in which this tile is
143
144
        * Oreturn the tunnel in which this tile is
```

```
*/
        Tunnel getTunnel();
147
148
149
         * Returns the {@link Tunnel.Exit} object associated with this tile info.
150
         * If the tile isn't in a tunnel, it returns null
151
152
         * @return the {@link Tunnel.Exit} object associated with this tile info or {@code
       null
         * Osee Tunnel.Exit
154
         */
155
        Tunnel.Exit getTunnelExit();
157
158
         * Returns {@code true} if this tile info is in a tunnel
159
         * Creturn {Code true} if this tile info is in a tunnel
161
162
        boolean isInATunnel();
163
165
         * Returns the room in which this tile is
166
167
         * Oreturn the room in which this tile is
168
169
        Room getRoom();
170
        /**
172
         * Returns {@code true} if this tile info is in a room
173
174
         * @return {@code true} if this tile info is in a room
176
        boolean isInARoom();
177
178
179
         * Oreturn {Ocode true} if this tile is marked
180
         * @see Mark
181
         * Osee MarkSystem
182
         */
        boolean isMarked();
184
185
        /**
186
         * Oparam dir the direction
         * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
188
         * @throws IndexOutOfBoundsException if this TileInfo is near the border of the board
189
       and
         * the direction point outside the board
190
191
        default TileInfo adjacent(Direction dir) {
192
            return getBoard().getAt(getX() + dir.dirX(), getY() + dir.dirY());
        }
194
195
196
         * Oparam dir the direction
197
         * @return the tile that is adjacent to this TileInfo in the {@link Direction} dir
198
         * or {@code null} if the adjacent tile is outside the board
199
200
        default TileInfo safeAdjacent(Direction dir) {
            return getBoard().safeGetAt(getX() + dir.dirX(), getY() + dir.dirY());
202
        }
203
204
        /**
```

```
* Returns the board in which this tile is
207
         * @return the board in which this tile is
208
         */
209
        Board getBoard();
210
211
        default int getIndex() {
212
            return getY() * getBoard().getWidth() + getX();
214
215
        /**
216
         * Represents the index of this crate in {@link
       fr.valax.sokoshell.solver.State#cratesIndices()}
         * array.
218
         * @return -1 if not set or the index of this crate in
219
                    {@link fr.valax.sokoshell.solver.State#cratesIndices()} array.
         */
221
        int getCrateIndex();
222
223
        TargetRemoteness getNearestTarget();
225
        TargetRemoteness[] getTargets();
226
227
        /**
         * @implNote If you replace index by TileInfo, you will need to modify
229
       MutableBoard#StaticTile.
         * If you are too lazy to do that, create an issue on github
230
231
        record TargetRemoteness(int index, int distance) implements
232
        → Comparable<TargetRemoteness> {
            @Override
234
            public int compareTo(TargetRemoteness other) {
235
                return this.distance - other.distance;
236
            }
237
            @Override
239
            public String toString() {
240
                return "TR[d=" + distance + ", i=" + index + "]";
242
        }
243
        // SETTERS //
246
247
        /**
248
         * If this was a floor, this is now a crate
         * 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
         * 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
         */
261
        void removeCrate();
262
263
        /**
```

```
* Sets the tile.
         * Oparam tile the new tile
266
         * @throws UnsupportedOperationException if the {@code setTile} operation isn't
267
         * supported by this TileInfo
268
         */
269
        void setTile(Tile tile);
270
271
         * Sets this tile as a dead tile or not
         * @throws UnsupportedOperationException if the {@code setDeadTile} operation isn't
274
         * supported by this TileInfo
275
        * @see MutableBoard#computeDeadTiles()
         */
277
        void setDeadTile(boolean deadTile);
278
279
        * Sets this tile as reachable or not by the player. It doesn't check if it's
281
       possible.
        * @throws UnsupportedOperationException if the {@code setReachable} operation isn't
282
         * supported by this TileInfo
         * @see MutableBoard#findReachableCases(int)
284
         */
285
        void setReachable(boolean reachable);
286
        /**
288
         * Sets the tunnel in which this tile is
289
         * @throws UnsupportedOperationException if the {@code setTunnel} operation isn't
         * supported by this TileInfo
291
         */
292
        void setTunnel(Tunnel tunnel);
293
295
         * Sets the {@link Tunnel.Exit} object associated with this tile info
296
         * @throws UnsupportedOperationException if the {@code setTunnelExit} operation isn't
297
         * supported by this TileInfo
298
         * @see Tunnel.Exit
299
         */
300
        void setTunnelExit(Tunnel.Exit tunnelExit);
301
        /**
303
         * Sets the room in which this tile is
304
         * @throws UnsupportedOperationException if the {@code setRoom} operation isn't
305
         * supported by this TileInfo
         */
307
        void setRoom(Room room);
308
309
        * Sets this tile as marked
311
         * @throws UnsupportedOperationException if the {@code mark} operation isn't
312
        * supported by this TileInfo
313
         * @see Mark
314
         * Osee MarkSystem
315
         */
316
        void mark();
317
        /**
319
        * Sets this tile as unmarked
320
        * @throws UnsupportedOperationException if the {@code unmark} operation isn't
        * supported by this TileInfo
322
         * @see Mark
323
         * Osee MarkSystem
324
         */
```

```
void unmark();
327
328
         * Sets this tile as marked or not
         * @throws UnsupportedOperationException if the {@code setMarked} operation isn't
330
         * supported by this TileInfo
331
         * Osee Mark
332
         * Osee MarkSystem
         */
334
        void setMarked(boolean marked);
335
336
        /**
         * Set the distance to every targets
338
         * Oparam targets distance to every targets
339
         * @throws UnsupportedOperationException if the {@code setTargets} operation isn't
340
         * supported by this TileInfo
341
342
        void setTargets(TargetRemoteness[] targets);
343
344
        /**
         * Set the nearest target
346
         * Oparam nearestTarget nearest target
347
         * @throws UnsupportedOperationException if the {@code setNearestTarget} operation
       isn't
         * supported by this TileInfo
349
         */
350
        void setNearestTarget(TargetRemoteness nearestTarget);
351
352
        /**
353
         * @see #getCrateIndex()
354
         */
        void setCrateIndex(int index);
356
357
```

### Tile

```
package fr.valax.sokoshell.solver.board.tiles;
2
3
   * Represents the content of a case of the board.
4
   */
   public enum Tile {
       FLOOR(false, false),
       WALL(true, false),
       CRATE(true, true),
       CRATE_ON_TARGET(true, true),
11
       TARGET(false, false);
12
13
14
       private final boolean solid;
15
16
       private final boolean crate;
17
       Tile(boolean solid, boolean crate) {
19
           this.solid = solid;
20
           this.crate = crate;
       }
22
23
24
        * Tells whether objects (i.e. player or crates) can move through the case or not.
        */
26
```

```
public boolean isSolid() {
    return solid;
}

/**

* Tells whether the case is occupied by a crate (on a target or not) or not.

*/

public boolean isCrate() {
    return crate;
}

}
```

## 1.4.2 mark

## HeavyweightMarkSystem

```
package fr.valax.sokoshell.solver.board.mark;
   import java.util.ArrayList;
3
   import java.util.List;
    * A heavyweight mark system contains a pointer to every mark associated with this system
   public class HeavyweightMarkSystem extends AbstractMarkSystem {
10
       protected final List<Mark> marks;
11
12
       public HeavyweightMarkSystem() {
           marks = new ArrayList<>();
14
15
16
       @Override
17
       public Mark newMark() {
18
           Mark m = super.newMark();
19
           marks.add(m);
20
           return m;
22
       }
23
       @Override
25
       public void reset() {
26
           mark = 0;
27
           for (Mark m : marks) {
29
                m.unmark();
30
           }
31
       }
32
33
```

## ${\bf Fixed Size Mark System}$

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

public class FixedSizeMarkSystem implements MarkSystem {

   protected final FMark[] marks;
   protected int mark;

public FixedSizeMarkSystem(int capacity) {
   marks = new FMark[capacity];
   for (int i = 0; i < capacity; i++) {</pre>
```

```
marks[i] = new FMark();
           }
12
       }
13
       public void mark(int i) {
15
           marks[i].mark();
16
17
       public boolean isMarked(int i) {
19
           return marks[i].isMarked();
20
       }
21
       @Override
23
       public Mark newMark() {
24
           throw new UnsupportedOperationException();
       @Override
28
       public void unmarkAll() {
29
           mark++;
31
           if (mark == 0) {
32
               reset();
           }
       }
35
36
       @Override
37
       public void reset() {
38
           mark = 0;
39
40
           for (FMark mark : marks) {
                mark.unmark();
42
43
       }
44
       @Override
       public int getMark() {
47
           return mark;
48
50
       private class FMark implements Mark {
51
52
           private int mark = 0;
54
           @Override
55
           public void mark() {
                mark = FixedSizeMarkSystem.this.mark;
58
59
           @Override
           public void unmark() {
61
                mark = FixedSizeMarkSystem.this.mark - 1;
62
63
           @Override
           public boolean isMarked() {
66
                return mark == FixedSizeMarkSystem.this.mark;
67
70
           public MarkSystem getMarkSystem() {
71
                return FixedSizeMarkSystem.this;
```

#### Mark

```
package fr.valax.sokoshell.solver.board.mark;
2
3
   * @see MarkSystem
    * @author PoulpoGaz
   */
   public interface Mark {
        * Marks the object. After this method is called, {@link #isMarked()}
10
        * will return {@code true}
        */
12
       void mark();
13
       /**
        * Un-marks the object. After this method is called, {@link #isMarked()}
16
        * will return {@code false}
17
        */
       void unmark();
19
20
21
        * Mark or not the object. After this method is called, {@link #isMarked()}
22
        * will return {@code marked}
24
       default void setMarked(boolean marked) {
25
           if (marked) {
               mark();
           } else {
28
               unmark();
29
           }
       }
31
32
33
       * Creturn true is the object is marked
35
       boolean isMarked();
36
37
       /**
        * Oreturn the {Olink MarkSystem} associated with this mark
39
40
       MarkSystem getMarkSystem();
41
   }
```

#### **DefaultMark**

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

public class DefaultMark implements Mark {

private final MarkSystem markSystem;
private int mark;

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

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

## MarkSystem

```
package fr.valax.sokoshell.solver.board.mark;
2
3
   * 
4
         A MarkSystem is used by dfs/bfs/others algorithm to avoid checking twice an object.
         With a MarkSystem, you don't need to unmark all visited objects
         {@link Mark} associated with this system can be created using {@link #newMark()}.
   * 
   * <h2>How it works</h2>
10
         A mark have a value, the same for a MarkSystem. A mark is marked if it value is
11

→ equals

         to the value of the MarkSystem. So, to unmark all mark, you just have to increase
          the MarkSystem's value.
13
   * 
14
   * @see Mark
15
   * @author PoulpoGaz
17
   public interface MarkSystem {
18
19
       /**
       * Create a new mark associated with this MarkSystem.
21
        * The mark is by default unmarked
22
       * Oreturn a new mark
23
       */
       Mark newMark();
25
26
       /**
       * Unmark all marks
       */
29
       void unmarkAll();
30
32
       * Set the 'selected' mark to 0 and unmark all Mark
33
34
       void reset();
36
```

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

# ${\bf AbstractMarkSystem}$

```
package fr.valax.sokoshell.solver.board.mark;
2
   * Contains the basic for all mark system
   public abstract class AbstractMarkSystem implements MarkSystem {
        * A mark is marked if it's value is equals to this field
9
10
       protected int mark;
11
12
       @Override
13
       public Mark newMark() {
           return new DefaultMark(this);
15
16
17
       @Override
       public void unmarkAll() {
19
           mark++;
20
^{21}
           if (mark == 0) {
               reset();
23
           }
24
       }
26
       @Override
27
       public abstract void reset();
28
29
       @Override
       public int getMark() {
31
           return mark;
32
       }
   }
```

## Tunnel

```
package fr.valax.sokoshell.solver.board;
  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
3
  import java.util.ArrayList;
  import java.util.List;
   * A tunnel is a zone of the board like this:
10
   * 
11
       $$$$$$
              $$$$$
         $$$$
14
            $$$$$$$
15
16 | *
```

```
*/
   public class Tunnel {
19
       // STATIC
20
21
       protected TileInfo start;
22
       protected TileInfo end;
23
       // the tile outside the tunnel adjacent to start
       protected TileInfo startOut;
26
27
       // the tile outside the tunnel adjacent to end
       protected TileInfo endOut;
29
       protected List<Room> rooms;
30
31
       // true if the tunnel can only be taken by the player
       protected boolean playerOnlyTunnel;
33
       protected boolean isOneway;
34
35
       // DYNAMIC
37
       protected boolean crateInside = false;
38
40
41
       public void createTunnelExits() {
42
           if (this.startOut != null) {
43
               Direction initDir = start.direction(startOut);
44
                create(start, initDir, startOut);
45
           }
46
           if (endOut != null) {
48
                Direction endDir = end.direction(endOut);
49
                create(end, endDir, endOut);
50
           }
51
       }
52
53
       private void create(TileInfo tile, Direction startDir, TileInfo startOut) {
54
           TileInfo t = tile;
56
           Direction nextDir = startDir.negate();
57
           while (true) {
               TileInfo next = t.adjacent(nextDir);
60
                if (next.isWall() || t.getTunnel() != this) {
61
                    break;
62
                }
64
               setExit(t, startDir, startOut);
65
                t = next;
67
           }
68
       }
69
70
       private void setExit(TileInfo tile, Direction dir, TileInfo out) {
71
           if (dir != null) {
72
               Exit exit = tile.getTunnelExit();
73
                if (exit == null) {
                    exit = new Exit();
76
                    tile.setTunnelExit(exit);
77
                }
```

```
79
                 switch (dir) {
80
                     case RIGHT -> exit.setRightExit(out);
81
                     case UP -> exit.setUpExit(out);
                     case DOWN -> exit.setDownExit(out);
83
                     case LEFT -> exit.setLeftExit(out);
84
                 }
85
            }
        }
87
88
        public void addRoom(Room room) {
89
            if (rooms == null) {
                 rooms = new ArrayList<>();
91
92
            rooms.add(room);
        }
95
        public List<Room> getRooms() {
96
            return rooms;
97
        }
99
        public TileInfo getStart() {
100
            return start;
101
103
        public void setStart(TileInfo start) {
104
            this.start = start;
105
106
107
        public TileInfo getEnd() {
108
            return end;
110
111
        public void setEnd(TileInfo end) {
112
            this.end = end;
113
114
115
        public TileInfo getStartOut() {
116
            return startOut;
118
119
        public void setStartOut(TileInfo startOut) {
120
            this.startOut = startOut;
        }
122
123
        public TileInfo getEndOut() {
124
            return endOut;
125
126
127
        public void setEndOut(TileInfo endOut) {
            this.endOut = endOut;
129
130
131
        public boolean isPlayerOnlyTunnel() {
132
            return playerOnlyTunnel;
134
135
        public void setPlayerOnlyTunnel(boolean playerOnlyTunnel) {
            this.playerOnlyTunnel = playerOnlyTunnel;
137
138
139
        public boolean crateInside() {
```

```
return crateInside;
        }
142
143
       public void setCrateInside(boolean crateInside) {
144
            this.crateInside = crateInside;
145
146
147
       public boolean isOneway() {
            return isOneway;
149
150
151
        public void setOneway(boolean oneway) {
            isOneway = oneway;
153
154
        Olverride
        public String toString() {
157
            if (startOut == null) {
158
                return
159
                    "closed - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
                         .formatted(start.getX(), start.getY(),
160
                                 end.getX(), end.getY(),
161
                                 endOut.getX(), endOut.getY(),
                                 playerOnlyTunnel, isOneway);
163
            } else if (endOut == null) {
164
                return
165
                   "(%d; %d) - (%d; %d) --> (%d; %d) - closed. only player? %s. one way? %s"
                         .formatted(startOut.getX(), startOut.getY(),
166
                                 start.getX(), start.getY(),
167
                                 end.getX(), end.getY(),
168
                                 playerOnlyTunnel, isOneway);
            } else {
170
                return
171
                    "(%d; %d) - (%d; %d) --> (%d; %d) - (%d; %d). only player? %s. one way? %s"
                         .formatted(startOut.getX(), startOut.getY(),
172
                                 start.getX(), start.getY(),
                                 end.getX(), end.getY(),
174
                                 endOut.getX(), endOut.getY(),
175
                                 playerOnlyTunnel, isOneway);
            }
177
178
179
         * Added to every tile that is inside a tunnel.
181
         * It contains for each direction where is the exit:
182
         * if you push a crate inside the tunnel to the left, the
183
         * method {@link #getExit(Direction)} wile return where you will
184
         * be after pushing the crate until you aren't outside the tunnel.
185
186
         * @implNote This object isn't immutable but is assumed as
187
         * immutable by
188
       MutableBoard.StaticBoard#linkTunnelsRoomsAndTileInfos(MutableBoard.StaticTile[][])
         */
189
       public static class Exit {
191
            private TileInfo leftExit;
192
            private TileInfo upExit;
193
            private TileInfo rightExit;
            private TileInfo downExit;
195
196
            public Exit() {
197
198
```

```
public Exit(TileInfo leftExit, TileInfo upExit, TileInfo rightExit, TileInfo
200
             → downExit) {
                 this.leftExit = leftExit;
201
                 this.upExit = upExit;
202
                 this.rightExit = rightExit;
203
                 this.downExit = downExit;
204
            }
206
            public TileInfo getExit(Direction dir) {
207
                 return switch (dir) {
208
                     case LEFT -> leftExit;
                     case UP -> upExit;
210
                     case RIGHT -> rightExit;
211
                     case DOWN -> downExit;
212
                 };
            }
214
215
            public TileInfo getLeftExit() {
216
                 return leftExit;
218
219
            private void setLeftExit(TileInfo leftExit) {
220
                 this.leftExit = leftExit;
222
223
            public TileInfo getUpExit() {
                 return upExit;
225
226
227
            private void setUpExit(TileInfo upExit) {
                 this.upExit = upExit;
229
230
231
            public TileInfo getRightExit() {
232
                 return rightExit;
234
235
            private void setRightExit(TileInfo rightExit) {
                 this.rightExit = rightExit;
237
238
239
            public TileInfo getDownExit() {
                 return downExit;
241
242
243
            private void setDownExit(TileInfo downExit) {
                 this.downExit = downExit;
245
246
        }
247
    }
248
```

# ImmutableBoard

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

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

import java.util.List;
```

```
9
10
   * Immutable implementation of {@link Board}.
11
12
   * This class extends {@link GenericBoard}. It internally uses {@link ImmutableTileInfo}
13
   \hookrightarrow to store the board content
   * in {@link GenericBoard#content}. As it is immutable, it implements the setters methods

→ always throws a

    * {@link UnsupportedOperationException} when such a method is called.
15
16
    * @see Board
17
    * @see GenericBoard
   * @see TileInfo
19
   */
20
   public class ImmutableBoard extends GenericBoard {
^{21}
       public ImmutableBoard(Tile[][] content, int width, int height) {
23
           super(width, height);
24
25
           this.content = new ImmutableTileInfo[height][width];
27
           for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    this.content[y][x] = new ImmutableTileInfo(this, content[y][x], x, y);
31
           }
32
       }
33
34
       public ImmutableBoard(Board other) {
35
           super(other.getWidth(), other.getHeight());
36
           this.content = new ImmutableTileInfo[height][width];
38
39
           for (int y = 0; y < height; y++) {
40
                for (int x = 0; x < width; x++) {
41
                    this.content[y][x] = new ImmutableTileInfo(other.getAt(x, y));
43
           }
44
       }
46
       // GETTERS //
47
       @Override
       public int getTargetCount() {
50
           return 0;
51
52
       @Override
54
       public List<Tunnel> getTunnels() {
55
           return null;
       }
57
58
       @Override
       public List<Room> getRooms() {
           return null;
61
62
63
       @Override
       public boolean isGoalRoomLevel() {
65
           return false;
66
       }
67
```

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

### Move

```
package fr.valax.sokoshell.solver.board;
   /**
3
   * An enumeration representing a move or a push in a solution. The {@code moveCrate} flag
   \hookrightarrow is needed to go back
   * in {@link fr.valax.sokoshell.commands.level.SolutionCommand}
6
    * DO NOT MODIFY ORDER OF VALUES WITHOUT REMAKING ALL SAVES
   public enum Move {
10
       LEFT("1", Direction.LEFT, false),
11
       UP("u", Direction.UP, false),
12
       DOWN("d", Direction.DOWN, false),
13
       RIGHT("r", Direction.RIGHT, false),
14
15
       LEFT_PUSH("L", Direction.LEFT, true),
16
       UP_PUSH("U", Direction.UP, true),
       RIGHT PUSH("R", Direction.RIGHT, true),
18
       DOWN_PUSH("D", Direction.DOWN, true);
19
       private final String shortName;
21
       private final Direction direction;
22
       private final boolean moveCrate;
23
       Move(String name, Direction direction, boolean moveCrate) {
25
           this.shortName = name;
26
           this.direction = direction;
27
           this.moveCrate = moveCrate;
       }
29
30
       public String shortName() {
31
           return shortName;
33
34
       public Direction direction() {
35
           return direction;
36
37
38
       public boolean moveCrate() {
           return moveCrate;
40
       }
41
42
       public static Move of(Direction dir, boolean moveCrate) {
43
           return switch (dir) {
44
               case LEFT -> moveCrate ? LEFT_PUSH : LEFT;
45
               case UP -> moveCrate ? UP_PUSH : UP;
46
               case DOWN -> moveCrate ? DOWN_PUSH : DOWN;
47
               case RIGHT -> moveCrate ? RIGHT_PUSH : RIGHT;
48
```

```
};
49
50
51
       public static Move of(String shortName) {
52
            for (Move move : Move.values()) {
53
                 if (move.shortName().equals(shortName)) {
54
                     return move;
55
                 }
            }
57
58
            return null;
59
       }
   }
61
```

## Room

```
package fr.valax.sokoshell.solver.board;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.ArrayList;
   import java.util.List;
   public class Room {
       protected boolean goalRoom;
10
       protected final List<TileInfo> tiles = new ArrayList<>();
12
       protected final List<TileInfo> targets = new ArrayList<>();
13
14
       protected List<Tunnel> tunnels;
16
17
        * Only computed if the level is a goal room level as defined by {@link
       Board#isGoalRoomLevel()}
19
       protected List<TileInfo> packingOrder;
20
       // dynamic
22
       // the index in packingOrder of the position of the next crate that will be pushed
23
       \hookrightarrow inside the room
       // negative if it is not possible because a crate isn't at the correct position
       // or if the room isn't a goal room
25
       protected int packingOrderIndex;
26
27
       public Room() {
29
30
       public void addTile(TileInfo tile) {
31
           tiles.add(tile);
33
           if (tile.isTarget()) {
34
               targets.add(tile);
           }
36
       }
37
38
       public List<TileInfo> getTiles() {
40
           return tiles;
41
42
43
       public List<TileInfo> getTargets() {
44
```

```
return targets;
45
       }
46
47
       public void addTunnel(Tunnel tunnel) {
49
           if (tunnels == null) {
50
                tunnels = new ArrayList<>();
51
           }
           tunnels.add(tunnel);
53
54
55
       public List<Tunnel> getTunnels() {
           return tunnels;
57
58
       public boolean isGoalRoom() {
61
           return goalRoom;
62
63
       public void setGoalRoom(boolean goalRoom) {
65
           this.goalRoom = goalRoom;
66
       public List<TileInfo> getPackingOrder() {
69
           return packingOrder;
70
       }
71
72
       public void setPackingOrder(List<TileInfo> packingOrder) {
73
           this.packingOrder = packingOrder;
       }
76
       public boolean isInPackingOrder(TileInfo tile) {
77
           return packingOrder != null && packingOrder.contains(tile);
78
       }
79
       public int getPackingOrderIndex() {
81
           return packingOrderIndex;
82
       public void setPackingOrderIndex(int packingOrderIndex) {
85
           this.packingOrderIndex = packingOrderIndex;
86
       }
   }
88
```

## MutableBoard

```
package fr.valax.sokoshell.solver.board;
  import fr.valax.sokoshell.SokoShell;
  import fr.valax.sokoshell.graphics.Surface;
  import fr.valax.sokoshell.solver.Corral;
  import fr.valax.sokoshell.solver.CorralDetector;
  import fr.valax.sokoshell.solver.State;
  import fr.valax.sokoshell.solver.board.mark.AbstractMarkSystem;
  import fr.valax.sokoshell.solver.board.mark.Mark;
  import fr.valax.sokoshell.solver.board.mark.MarkSystem;
  import fr.valax.sokoshell.solver.board.tiles.GenericTileInfo;
11
  import fr.valax.sokoshell.solver.board.tiles.MutableTileInfo;
12
  import fr.valax.sokoshell.solver.board.tiles.Tile;
13
  import fr.valax.sokoshell.solver.board.tiles.TileInfo;
  import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
```

```
import fr.valax.sokoshell.solver.pathfinder.CratePlayerAStar;
   import fr.valax.sokoshell.solver.pathfinder.PlayerAStar;
17
18
   import java.util.*;
   import java.util.function.Consumer;
21
22
    * Mutable implementation of {@link Board}.
24
25
   * This class extends {@link GenericBoard} by defining all the setters methods. It
   → internally uses {@link MutableTileInfo} to store the board content
    * in {@link GenericBoard#content}.
27
28
    * @see Board
29
   * @see GenericBoard
    * @see MutableTileInfo
31
32
   @SuppressWarnings("ForLoopReplaceableByForEach")
   public class MutableBoard extends GenericBoard {
35
       private final MarkSystem markSystem = newMarkSystem(TileInfo::unmark);
36
       private final MarkSystem reachableMarkSystem = newMarkSystem((t) ->
37

    t.setReachable(false));
38
       private int targetCount;
39
       /**
41
        * Tiles that can be 'target' or 'floor'
42
43
       private TileInfo[] floors;
45
       private final List<Tunnel> tunnels = new ArrayList<>();
46
       private final List<Room> rooms = new ArrayList<>();
47
       /**
50
        * True if all rooms are goal room with only one entrance
51
       private boolean isGoalRoomLevel;
53
54
       private PlayerAStar playerAStar;
       private CrateAStar crateAStar;
       private CratePlayerAStar cratePlayerAStar;
57
58
       private final CorralDetector corralDetector;
59
       private StaticBoard staticBoard;
61
62
       /**
        * Creates a SolverBoard with the specified width, height and tiles
64
65
        * Oparam content a rectangular matrix of size width * height. The first index is for
66
      the rows
                         and the second for the columns
67
        * @param width board width
68
        * Oparam height board height
69
       public MutableBoard(Tile[][] content, int width, int height) {
71
           super(width, height);
72
73
           this.content = new TileInfo[height][width];
```

```
for (int y = 0; y < height; y++) {
75
                for (int x = 0; x < width; x++) {
76
                    this.content[y][x] = new MutableTileInfo(this, content[y][x], x, y);
77
            }
80
            corralDetector = new CorralDetector(this);
       }
83
       public MutableBoard(int width, int height) {
84
            super(width, height);
85
            this.content = new TileInfo[height][width];
87
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    this.content[y][x] = new MutableTileInfo(this, Tile.FLOOR, x, y);
91
            }
92
93
            corralDetector = new CorralDetector(this);
       }
95
96
        * Creates a copy of 'other'. It doesn't copy solver information
99
         * Oparam other the board to copy
100
101
       public MutableBoard(Board other) {
102
            this(other, false);
103
104
       public MutableBoard(Board other, boolean copyStatic) {
106
            super(other.getWidth(), other.getHeight());
107
108
            content = new TileInfo[height][width];
            for (int y = 0; y < height; y++) {
110
                for (int x = 0; x < width; x++) {
111
                    content[y][x] = new MutableTileInfo(this, other.getAt(x, y));
112
                }
            }
114
115
            corralDetector = new CorralDetector(this);
            if (copyStatic) {
118
                copyStaticInformation(other);
119
            }
120
       }
121
122
       private void copyStaticInformation(Board other) {
123
            // map room in other board and in this board
            Map<Room, Room> roomMap = new HashMap<>(rooms.size());
125
            Map<Tunnel, Tunnel> tunnelMap = new HashMap<>(rooms.size());
126
127
            // copy tunnels, rooms
            for (Room room : other.getRooms()) {
                Room copy = copyRoom(room);
130
                roomMap.put(room, copy);
131
                rooms.add(copy);
            }
133
            for (Tunnel tunnel : other.getTunnels()) {
134
                Tunnel copy = copyTunnel(tunnel);
135
136
                tunnelMap.put(tunnel, copy);
```

```
tunnels.add(copy);
137
            }
138
139
            // copy tile info
            for (int y = 0; y < height; y++) {
141
                for (int x = 0; x < width; x++) {
142
                    TileInfo otherTile = other.getAt(x, y);
143
                    TileInfo tile = content[y][x];
                    tile.setDeadTile(otherTile.isDeadTile());
145
146
                     if (tile.getTargets() != null) {
147
                         tile.setTargets(Arrays.copyOf(tile.getTargets(),

→ tile.getTargets().length));
149
                    tile.setNearestTarget(otherTile.getNearestTarget());
                    tile.setTunnel(tunnelMap.get(otherTile.getTunnel()));
152
                    tile.setRoom(roomMap.get(otherTile.getRoom()));
153
                    if (otherTile.getTunnelExit() != null) {
154
                         tile.setTunnelExit(otherTile.getTunnelExit()); // it is immutable !
156
                }
157
            }
            // link rooms and tunnels
160
            for (Tunnel tunnel : other.getTunnels()) {
161
                Tunnel newTunnel = tunnelMap.get(tunnel);
162
                for (Room room : other.getRooms()) {
163
                    Room newRoom = roomMap.get(room);
164
                    newTunnel.addRoom(newRoom);
165
                    newRoom.addTunnel(newTunnel);
                }
167
            }
168
        }
169
170
       private Room copyRoom(Room room) {
171
            Room newRoom = new Room();
172
            newRoom.setGoalRoom(room.isGoalRoom());
173
            for (TileInfo t : room.getTiles()) {
175
                newRoom.addTile(getAt(t.getIndex()));
176
            }
177
            if (room.getPackingOrder() != null) {
                List<TileInfo> packingOrder = new ArrayList<>();
179
                for (TileInfo t : room.getPackingOrder()) {
180
                    packingOrder.add(getAt(t.getIndex()));
182
                newRoom.setPackingOrder(packingOrder);
183
            }
184
            return newRoom;
186
        }
187
       private Tunnel copyTunnel(Tunnel tunnel) {
            Tunnel newTunnel = new Tunnel();
190
191
            newTunnel.setStart(getAt(tunnel.getStart().getIndex()));
192
            newTunnel.setEnd(getAt(tunnel.getEnd().getIndex()));
194
            if (tunnel.getStartOut() != null) {
195
                newTunnel.setStartOut(getAt(tunnel.getStartOut().getIndex()));
196
            }
197
```

```
if (tunnel.getEndOut() != null) {
198
                 newTunnel.setEndOut(getAt(tunnel.getEndOut().getIndex()));
199
            }
200
            newTunnel.setPlayerOnlyTunnel(tunnel.isPlayerOnlyTunnel());
201
            newTunnel.setOneway(tunnel.isOneway());
202
203
            return newTunnel;
204
        }
206
207
         * Apply the consumer on every tile info
208
         * Oparam consumer the consumer to apply
210
         */
211
        public void forEach(Consumer<TileInfo> consumer) {
212
            for (int y = 0; y < height; y++) {
                 for (int x = 0; x < width; x++) {
214
                     consumer.accept(content[y][x]);
215
216
            }
        }
218
219
220
         * Set at tile at the specified index. The index will be converted to
         * cartesian coordinate with {@link #getX(int)} and {@link #getY(int)}
222
223
         * Oparam index index in the board
         * Oparam tile the new tile
225
         * @throws IndexOutOfBoundsException if the index lead to a position outside the board
226
227
        public void setAt(int index, Tile tile) {

    content[getY(index)][getX(index)].setTile(tile); }

229
        /**
230
         * Set at tile at (x, y)
231
232
         * Oparam x x position in the board
233
         * Oparam y y position in the board
234
         * @throws IndexOutOfBoundsException if the position is outside the board
236
        public void setAt(int x, int y, Tile tile) {
237
            content[y][x].setTile(tile);
238
        }
240
241
         * Puts the crates of the given state in the content array.
242
243
         * Oparam state The state with the crates
244
245
        public void addStateCrates(State state) {
246
            int[] cratesIndices = state.cratesIndices();
247
            for (int j = 0; j < cratesIndices.length; j++) {</pre>
248
                 int i = cratesIndices[j];
249
                TileInfo crate = getAt(i);
250
                 crate.setCrateIndex(j);
                 crate.addCrate();
252
            }
253
        }
255
256
257
         * Removes the crates of the given state from the content array.
258
```

```
* Oparam state The state with the crates
260
       public void removeStateCrates(State state) {
261
           for (int i : state.cratesIndices()) {
               TileInfo crate = getAt(i);
263
               crate.setCrateIndex(-1);
264
               crate.removeCrate();
265
           }
       }
267
268
269
        * Puts the crates of the given state in the content array.
        * If a crate is outside the board, it doesn't throw an {@link
271
      IndexOutOfBoundsException}
272
        * Oparam state The state with the crates
274
       public void safeAddStateCrates(State state) {
275
           for (int i : state.cratesIndices()) {
276
               TileInfo info = safeGetAt(i);
278
               if (info != null) {
279
                   info.addCrate();
280
               }
           }
282
       }
283
285
        * Removes the crates of the given state from the content array.
286
        * If a crate is outside the board, it doesn't throw an {@link
287
      IndexOutOfBoundsException}
288
        * @param state The state with the crates
289
290
       public void safeRemoveStateCrates(State state) {
291
           for (int i : state.cratesIndices()) {
292
               TileInfo info = safeGetAt(i);
293
294
               if (info != null) {
                   info.removeCrate();
296
               }
297
           }
298
       }
300
       301
       // *
            Methods used by solvers
302
       // * You need to call #initForSolver() first *
       304
305
       /**
        * Initialize the board for solving:
307
        * 
308
              compute floor tiles: an array containing all non-wall tile
309
              compute {@linkplain #computeDeadTiles() dead tiles}
310
              find {@linkplain #findTunnels() tunnels}
312
        * <strong>The board must have no crate inside</strong>
313
        * @see Tunnel
        */
315
       public void initForSolver() {
316
           playerAStar = new PlayerAStar(this);
317
           crateAStar = new CrateAStar(this);
```

```
cratePlayerAStar = new CratePlayerAStar(this);
320
            computeFloors();
321
            computeDeadTiles();
322
            findTunnels();
323
            findRooms();
324
            removeUselessTunnels();
325
            finishComputingTunnels();
            tryComputePackingOrder();
327
            computeTileToTargetsDistances();
328
329
            // we must compute the static board here
            // this is the unique point where the board
331
            // information are guaranteed to be true.
332
            // For example, the freeze deadlock detector
333
            // places wall on the map but this object
            // has no information about this.
335
            staticBoard = new StaticBoard();
336
        }
337
339
         * Creates or recreates the floor array. It is an array containing all tile info
340
         * that are not a wall
341
        public void computeFloors() {
343
            int nFloor = 0;
344
            for (int y = 0; y < height; y++) {
345
                 for (int x = 0; x < width; x++) {
346
                     TileInfo t = getAt(x, y);
347
348
                     if (!t.isSolid() || t.isCrate()) {
                         nFloor++;
350
351
                 }
352
            }
353
354
            this.floors = new TileInfo[nFloor];
355
            int i = 0;
356
            for (int y = 0; y < height; y++) {
                 for (int x = 0; x < width; x++) {
358
                     if (!this.content[y][x].isSolid() || this.content[y][x].isCrate()) {
359
                         this.floors[i] = this.content[y][x];
360
                         i++;
                     }
362
                 }
363
            }
364
        }
366
367
         * Apply the consumer on every tile info except walls
369
         * @param consumer the consumer to apply
370
         */
371
        public void forEachNotWall(Consumer<TileInfo> consumer) {
372
            for (TileInfo floor : floors) {
                 consumer.accept(floor);
374
            }
375
        }
376
377
        public void computeTunnelStatus(State state) {
378
            for (int i = 0; i < tunnels.size(); i++) {</pre>
379
                 tunnels.get(i).setCrateInside(false);
```

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

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

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

```
if (pushDir1 == null) { // all adjacents tiles are wall, ie init is alone, nerver
561
                happen see LevelBuilder
                 return null;
562
            } else if (pushDir2 == null) {
563
564
                     We are in this case:
565
                       1$1
566
                      $| |$
568
569
                 Tunnel tunnel = new Tunnel();
570
                 tunnel.setStart(init);
                 tunnel.setEnd(init);
572
                 tunnel.setEndOut(init.adjacent(pushDir1));
573
                 init.setTunnel(tunnel);
                 growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
576
                 return tunnel;
577
            } else {
578
                 /*
                     Either:
580
                     # | |#
581
                     Either:
                      |#|
583
                     #|
584
585
                 boolean onlyPlayer = false;
587
                 if (pushDir1.negate() != pushDir2) {
588
589
                          First case:
                            |#|
591
                           #|i|
592
                            | |#
593
                          if init is like this, then this is a tunnel and a crate
594
                          mustn't be pushed inside.
595
596
                          Second case:
597
                            |#|
                           #|i|
599
600
                          ie not tunnel
601
602
                     if (init.adjacent(pushDir1).adjacent(pushDir2).isSolid()) {
603
                          onlyPlayer = true;
604
                     } else {
605
                          return null;
606
                     }
607
                 }
608
609
                 Tunnel tunnel = new Tunnel();
610
                 tunnel.setEnd(init);
611
                 tunnel.setEndOut(init.adjacent(pushDir1));
612
                 tunnel.setPlayerOnlyTunnel(onlyPlayer);
613
                 init.setTunnel(tunnel);
615
                 growTunnel(tunnel, init.adjacent(pushDir1), pushDir1);
616
617
                 tunnel.setStart(tunnel.getEnd());
                 tunnel.setStartOut(tunnel.getEndOut());
618
                 growTunnel(tunnel, init.adjacent(pushDir2), pushDir2);
619
620
621
                 return tunnel;
```

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

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

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

```
t1.setEnd(t2.getEnd());
807
                     t1.setEndOut(t2.getEndOut());
808
                 } else {
809
                     t1.setEnd(t2.getStart());
                     t1.setEndOut(t2.getStartOut());
811
                 }
812
            }
813
            forEachNotWall((t) -> {
815
                 if (t.getTunnel() == t2) {
816
                     t.setTunnel(t1);
817
                 }
            });
819
820
            toAdd.setRoom(null);
            toAdd.setTunnel(t1);
            t1.setPlayerOnlyTunnel(t1.isPlayerOnlyTunnel() || t2.isPlayerOnlyTunnel());
823
            t1.rooms.remove(room);
824
            t2.rooms.remove(room);
825
            for (Room r : t2.rooms) {
827
                 r.tunnels.remove(t2);
828
                 r.tunnels.add(t1);
            }
831
            t1.rooms.addAll(t2.rooms);
832
        }
833
834
        private void finishComputingTunnels() {
835
            for (int i = 0; i < tunnels.size(); i++) {</pre>
836
                 Tunnel tunnel = tunnels.get(i);
838
                 // compute tunnel exits
839
                 tunnel.createTunnelExits();
840
841
                 // compute oneway property
                 if (tunnel.getStartOut() == null || tunnel.getEndOut() == null) {
843
                     tunnel.setOneway(true);
844
                 } else {
                     tunnel.getStart().addCrate();
846
                     corralDetector.findCorral(this, tunnel.getStartOut().getX(),
847

    tunnel.getStartOut().getY());
                     tunnel.getStart().removeCrate();
849
                     tunnel.setOneway(!tunnel.getEndOut().isReachable());
850
                 }
            }
        }
853
854
         * Compute packing order. No crate should be on the board
856
857
        public void tryComputePackingOrder() {
            isGoalRoomLevel = rooms.size() > 1;
            if (!isGoalRoomLevel) {
861
                 return:
862
            }
864
            for (int i = 0; i < rooms.size(); i++) {</pre>
865
                 Room r = rooms.get(i);
866
                 if (r.isGoalRoom() && r.getTunnels().size() != 1) {
867
```

```
isGoalRoomLevel = false;
868
                     break;
869
                 }
870
            }
871
872
            if (isGoalRoomLevel) {
873
                 for (Room r : rooms) {
874
                     if (r.isGoalRoom() && !computePackingOrder(r)) {
                          isGoalRoomLevel = false; // failed to compute packing order for a
876

→ room...

                         break;
877
                     }
                 }
879
            }
880
        }
883
         * The room must have only one entrance and a packing room
884
         * @param room a room
885
         */
        private boolean computePackingOrder(Room room) {
887
            markSystem.unmarkAll();
888
            Tunnel tunnel = room.getTunnels().get(0);
890
            TileInfo entrance;
891
            TileInfo inRoom;
892
            if (tunnel.getStartOut() != null && tunnel.getStartOut().getRoom() == room) {
                 entrance = tunnel.getStart();
894
                 inRoom = tunnel.getStartOut();
895
            } else {
896
                 entrance = tunnel.getEnd();
                 inRoom = tunnel.getEndOut();
898
899
900
            List<TileInfo> targets = room.getTargets();
901
            for (TileInfo t : targets) {
902
                 t.addCrate();
903
            }
904
            List<TileInfo> packingOrder = new ArrayList<>();
906
907
908
            List<TileInfo> frontier = new ArrayList<>();
            List<TileInfo> newFrontier = new ArrayList<>();
910
            frontier.add(entrance);
911
912
            List<TileInfo> accessibleCrates = new ArrayList<>();
            findAccessibleCrates(frontier, newFrontier, accessibleCrates);
914
915
            while (!accessibleCrates.isEmpty()) {
                 boolean hasChanged = false;
917
918
                 for (int i = 0; i < accessibleCrates.size(); i++) {</pre>
919
                     TileInfo crate = accessibleCrates.get(i);
920
                     crate.removeCrate();
                     inRoom.addCrate();
922
923
                     if (crateAStar.hasPath(entrance, null, inRoom, crate)) {
                         accessibleCrates.remove(i);
925
                         i--;
926
                         crate.unmark();
927
                         crate.removeCrate();
```

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

```
}
990
991
             if (!newFrontier.isEmpty()) {
992
                  findAccessibleCrates(newFrontier, lastFrontier, out);
             } else {
994
                  lastFrontier.clear();
995
996
         }
998
        private void computeTileToTargetsDistances() {
999
1000
             List<Integer> targetIndices = new ArrayList<>();
1001
1002
             targetCount = 0;
1003
             for (int y = 0; y < height; y++) {
1004
                  for (int x = 0; x < width; x++) {
                      if (this.content[y][x].isTarget() || this.content[y][x].isCrateOnTarget())
1006
                          {
                           targetCount++;
1007
                           targetIndices.add(getIndex(x, y));
1008
                      }
1009
                  }
1010
             }
1011
1012
             for (int y = 0; y < height; y++) {
1013
                  for (int x = 0; x < width; x++) {
1014
1015
                      final TileInfo t = getAt(x, y);
1016
1017
                      int minDistToTarget = Integer.MAX_VALUE;
1018
                      int minDistToTargetIndex = -1;
1020
                      getAt(x, y).setTargets(new
1021
                          TileInfo.TargetRemoteness[targetIndices.size()]);
1022
                      for (int j = 0; j < targetIndices.size(); j++) {</pre>
1023
1024
                          final int targetIndex = targetIndices.get(j);
1025
                          final int d = (t.isFloor() || t.isTarget()
                                           ? playerAStar.findPath(t, getAt(targetIndex), null,
1027
                                           → null).getDist()
                                           : 0);
1028
1029
1030
                           if (d < minDistToTarget) {</pre>
1031
                               minDistToTarget = d;
1032
                               minDistToTargetIndex = j;
1033
                           }
1034
1035
                           getAt(x, y).getTargets()[j] = new

→ TileInfo.TargetRemoteness(targetIndex, d);

                      }
1037
                      Arrays.sort(getAt(x, y).getTargets());
1038
                      getAt(x, y).setNearestTarget(new
1039
                          TileInfo.TargetRemoteness(minDistToTargetIndex, minDistToTarget));
                  }
1040
             }
1041
         }
1042
1043
1044
1045
1046
```

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

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

```
}
1168
        public CrateAStar getCrateAStar() {
1169
             return crateAStar;
1170
         }
1171
1172
        public CratePlayerAStar getCratePlayerAStar() {
1173
             return cratePlayerAStar;
1175
1176
         @Override
1177
        public Corral getCorral(TileInfo tile) {
             return corralDetector.findCorral(tile);
1179
1180
1181
         Olverride
        public CorralDetector getCorralDetector() {
1183
             return corralDetector;
1184
1185
1187
          * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking
1188
        twice a tile
1189
          * Oreturn a mark system
1190
         * @see MarkSystem
1191
1192
        public MarkSystem getMarkSystem() {
1193
             return markSystem;
1194
1195
1197
         * Returns the {@linkplain MarkSystem mark system} used by the {@link
1198
        #findReachableCases(int)} algorithm
1199
         * @return the reachable mark system
1200
          * @see MarkSystem
1201
          */
1202
        public MarkSystem getReachableMarkSystem() {
             return reachableMarkSystem;
1204
1205
1206
          * Creates a {@linkplain MarkSystem mark system} that apply the specified reset
1208
          * consumer to every <strong>non-wall</strong> {@linkplain TileInfo tile info}
1209
          * that are in this {@linkplain Board board}.
1210
1211
          * Oparam reset the reset function
1212
          * Oreturn a new MarkSystem
1213
         * @see MarkSystem
1214
          * @see Mark
1215
          */
1216
        private MarkSystem newMarkSystem(Consumer<TileInfo> reset) {
1217
             return new AbstractMarkSystem() {
1218
                 @Override
                 public void reset() {
1220
                      mark = 0;
1221
                      forEachNotWall(reset);
                 }
1223
             };
1224
        }
1225
1226
```

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

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

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

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

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

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

#### Direction

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

```
* @return the direction rotated by 90°
44
       public Direction left() {
45
           return switch (this) {
               case DOWN -> RIGHT;
47
               case LEFT -> DOWN;
48
               case UP -> LEFT;
49
               case RIGHT -> UP;
           };
51
       }
52
53
        * Rotate the rotation by -90°. For {@link Direction#UP} it returns {@link
55
   → Direction#RIGHT}
56
        * @return the direction rotated by -90°
57
58
       public Direction right() {
59
           return switch (this) {
               case DOWN -> LEFT;
               case LEFT -> UP;
62
               case UP -> RIGHT;
63
               case RIGHT -> DOWN;
65
           };
       }
66
67
        * @return The opposite direction (e.g for {@link Direction#LEFT} it returns {@link
   → Direction#LEFT} etc.)
        */
70
       public Direction negate() {
71
           return switch (this) {
72
               case DOWN -> UP;
73
               case UP -> DOWN;
74
               case LEFT -> RIGHT;
               case RIGHT -> LEFT;
           };
77
       }
78
80
        * Creates a direction from two coordinates.
81
        * Oparam dirX If negative, returns {Olink Direction#LEFT}, otherwise returns {Olink
   → Direction#RIGHT}
       * @param dirY If negative, return {@link Direction#UP}, otherwise returns {@link
83
   → Direction#DOWN}
        * @return the direction
84
       public static Direction of(int dirX, int dirY) {
86
           if (dirX == 0 && dirY == 0) {
87
               throw new IllegalArgumentException("(0,0) is not a direction");
           } else if (dirX == 0) {
89
               if (dirY < 0) {
90
                    return UP;
91
               } else {
92
                    return DOWN;
93
               }
94
           } else if (dirX < 0) {
95
               return LEFT;
           } else {
               return RIGHT;
98
           }
99
       }
```

.01

### GenericBoard

```
package fr.valax.sokoshell.solver.board;
   import fr.valax.sokoshell.solver.Corral;
   import fr.valax.sokoshell.solver.CorralDetector;
   import fr.valax.sokoshell.solver.State;
   import fr.valax.sokoshell.solver.board.tiles.Tile;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.function.Consumer;
9
10
11
   * A {@code package-private} class meant to be use as a base class for {@link Board}
12
   \hookrightarrow implementations.
   * It defines all read-only methods, as well as a way to store the tiles. It is
   \hookrightarrow essentially a 2D-array of
    * {@link TileInfo}, the indices being the y and x coordinates (i.e. {@code content[y][x]}
14
   \rightarrow is the tile at (x;y)).
15
    * @see Board
16
    * @see TileInfo
17
    */
18
   public abstract class GenericBoard implements Board {
       protected final int width;
21
22
       protected final int height;
23
       protected TileInfo[][] content;
25
26
       public GenericBoard(int width, int height) {
           this.width = width;
28
           this.height = height;
29
       }
30
       @SuppressWarnings("CopyConstructorMissesField")
32
       public GenericBoard(Board other) {
33
           this(other.getWidth(), other.getHeight());
       }
36
       @Override
37
       public int getWidth() { return width; }
40
       public int getHeight() { return height; }
41
       @Override
43
       public int getY(int index) { return index / width; }
44
45
       @Override
       public int getX(int index) { return index % width; }
47
48
49
       public int getIndex(int x, int y) { return y * width + x; }
51
       @Override
52
       public TileInfo getAt(int index) {
53
           return content[getY(index)][getX(index)];
       }
55
```

```
56
        @Override
57
       public TileInfo getAt(int x, int y) {
58
            return content[y][x];
       }
60
61
62
       // SETTERS: throw UnsupportedOperationException as this object is immutable //
        @Override
64
       public void forEach(Consumer<TileInfo> consumer) {
65
           throw new UnsupportedOperationException("Board is immutable");
       }
68
       @Override
69
       public void setAt(int index, Tile tile) {
            throw new UnsupportedOperationException("Board is immutable");
72
73
       @Override
74
       public void setAt(int x, int y, Tile tile) {
            throw new UnsupportedOperationException("Board is immutable");
76
       @Override
       public void addStateCrates(State state) {
80
            throw new UnsupportedOperationException("Board is immutable");
81
       }
       @Override
       public void removeStateCrates(State state) {
85
            throw new UnsupportedOperationException("Board is immutable");
87
88
       @Override
89
       public void safeAddStateCrates(State state) {
            throw new UnsupportedOperationException("Board is immutable");
       }
92
       @Override
       public void safeRemoveStateCrates(State state) {
95
           throw new UnsupportedOperationException("Board is immutable");
96
       }
97
        // Solver-used methods: throw UnsupportedOperationException as this object is (for
99
        \rightarrow now) not to be used by solvers //
100
       @Override
101
       public void initForSolver() {
102
            throw new UnsupportedOperationException("Board is not intended for solvers");
103
        }
105
       @Override
106
       public void computeFloors() {
107
            throw new UnsupportedOperationException("Board is not intended for solvers");
       }
110
       @Override
111
       public void forEachNotWall(Consumer<TileInfo> consumer) {
            throw new UnsupportedOperationException("Board is not intended for solvers");
113
114
115
        @Override
```

```
public void computeTunnelStatus(State state) {
            throw new UnsupportedOperationException("Board is not intended for solvers");
118
        }
119
120
        @Override
121
       public void computePackingOrderProgress(State state) {
122
            throw new UnsupportedOperationException("Board is not intended for solvers");
123
125
        @Override
126
        public void computeDeadTiles() {
127
            throw new UnsupportedOperationException("Board is not intended for solvers");
129
130
        Onverride
131
       public void findTunnels() {
            throw new UnsupportedOperationException("Board is not intended for solvers");
133
134
135
        @Override
       public void findRooms() {
137
            throw new UnsupportedOperationException("Board is not intended for solvers");
138
        }
        @Override
141
       public void tryComputePackingOrder() {
142
            throw new UnsupportedOperationException("Board is not intended for solvers");
        }
144
145
        @Override
146
        public void findReachableCases(int playerPos) {
            throw new UnsupportedOperationException("Board is not intended for solvers");
148
149
150
        @Override
151
        public int topLeftReachablePosition(TileInfo crate, TileInfo crateDest) {
152
            throw new UnsupportedOperationException("Board is not intended for solvers");
153
        }
154
        @Override
156
       public Corral getCorral(TileInfo tile) {
157
            return null;
160
        @Override
161
       public CorralDetector getCorralDetector() {
162
            return null;
163
164
165
```

### Board

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

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

```
import java.util.List;
   import java.util.function.Consumer;
12
13
   * Represents the Sokoban board.<br />
   * This interface defines getters setters for the properties of a Sokoban board, e.g. the
16
   \hookrightarrow width, the height etc.
   * Implementations of this interface are meant to be used with a {@link TileInfo}
   \rightarrow implementation.
   * This class also defines static and dynamic analysis of the Sokoban board, for instance
18
   \rightarrow for solving purposes.
   * Such properties are the following:
    * 
20
         Static
21
         <111>
22
             >Dead positions: cases that make the level unsolvable when a crate is pushed
      on them
         24
         Dynamic
25
         Reachable cases: cases that the player can reach according to his
27
   → position
         </117>
28
    * 
30
    * @see TileInfo
31
32
   public interface Board {
33
34
       int MINIMUM_WIDTH = 5;
35
       int MINIMUM_HEIGHT = 5;
37
       // GETTERS //
38
39
       /**
       * Returns the width of the board
42
        * @return the width of the board
43
        */
       int getWidth();
45
46
       /**
47
       * Returns the height of the board
49
        * @return the height of the board
50
        */
51
       int getHeight();
52
53
54
       * Returns the number of target i.e. tiles on which a crate has to be pushed to solve
      the level on the board
56
       * @return the number of target i.e. tiles on which a crate has to be pushed to solve
57
   \hookrightarrow the level on the board
58
       int getTargetCount();
59
60
       /**
       * Convert an index to a position on the y-axis
63
       * Oparam index the index to convert
64
        * @return the converted position
```

```
int getY(int index);
67
68
        * Convert an index to a position on the x-axis
71
         * Oparam index the index to convert
72
         * Oreturn the converted position
74
        int getX(int index);
75
76
        /**
        * Convert a (x;y) position to an index
78
79
        * @param x Coordinate on x-axis
80
        * Oparam y Coordinate on y-axis
         * @return the converted index
82
83
        int getIndex(int x, int y);
84
86
        * Returns the {@link TileInfo} at the specific index
87
        * Oparam index the index of the {Olink TileInfo}
         * Oreturn the TileInfo at the specific index
90
        * @throws IndexOutOfBoundsException if the index lead to a position outside the board
91
        * Osee #getX(int)
        * @see #getY(int)
        * @see #safeGetAt(int)
94
         */
95
       TileInfo getAt(int index);
97
        /**
98
        * Returns the {@link TileInfo} at the specific index
99
        * Oparam index the index of the {Olink TileInfo}
         * Oreturn the TileInfo at the specific index or {Ocode null}
102
        * if the index represent a position outside the board
103
        * @see #getX(int)
         * Osee #getY(int)
105
106
       default TileInfo safeGetAt(int index) {
            int x = getX(index);
            int y = getY(index);
109
110
            if (caseExists(x, y)) {
111
                return getAt(x, y);
            } else {
113
                return null;
114
            }
115
       }
116
117
118
        * Returns the {@link TileInfo} at the specific position
        * Oparam x x the of the tile
121
        * Oparam y y the of the tile
122
        * @return the TileInfo at the specific coordinate
        * @throws IndexOutOfBoundsException if the position is outside the board
        * @see #safeGetAt(int, int)
125
         */
126
       TileInfo getAt(int x, int y);
```

```
/**
129
         * Returns the {@link TileInfo} at the specific position
130
131
         * Oparam x x the of the tile
132
         * Oparam y y the of the tile
133
         * Oreturn the TileInfo at the specific index or {Ocode null}
134
         * if the index represent a position outside the board
         * Osee #getX(int)
136
         * @see #getY(int)
137
138
        default TileInfo safeGetAt(int x, int y) {
            if (caseExists(x, y)) {
140
                return getAt(x, y);
141
            } else {
142
                return null;
            }
144
        }
145
146
         * Tells whether the case at (x,y) exists or not (i.e. if the case is in the board)
148
149
         * Oparam x x-coordinate
         * Oparam y y-coordinate
         * @return {@code true} if the case exists, {@code false} otherwise
152
153
        default boolean caseExists(int x, int y) {
            return (0 <= x && x < getWidth()) && (0 <= y && y < getHeight());
155
        }
156
157
         * Same than caseExists(x, y) but with an index
159
160
         * Oparam index index of the case
161
         * Oreturn {Ocode true} if the case exists, {Ocode false} otherwise
162
         * @see #caseExists(int, int)
163
         */
164
        default boolean caseExists(int index) {
165
            return caseExists(getX(index), getY(index));
        }
167
168
169
         * Tells whether the tile at the given coordinates is empty or not.
171
         * Oparam x x coordinate of the case
172
         * Oparam y y coordinate of the case
173
         * @return {@code true} if empty, {@code false} otherwise
174
         */
175
        default boolean isTileEmpty(int x, int y) {
176
            TileInfo t = getAt(x, y);
            return !t.isSolid();
178
        }
179
180
        /**
181
         * Checks if the board is solved (i.e. all the crates are on a target). 
         * <strong>The crates MUSTileInfo have been put on the board for this function to work
183
       as expected.</strong>
         * @return {@code true} if the board is completed, false otherwise
185
         */
186
        default boolean isCompletedWith(State s) {
187
            for (int i : s.cratesIndices()) {
```

```
if (!getAt(i).isCrateOnTarget()) {
                     return false;
190
191
            }
            return true;
193
        }
194
195
         * Checks if the board is completed (i.e. all the crates are on a target)
197
198
         * @return true if completed, false otherwise
199
         */
        default boolean isCompleted() {
201
            for (int y = 0; y < getHeight(); y++) {
202
                 for (int x = 0; x < getWidth(); x++) {
203
                     if (getAt(x, y).isCrate()) {
204
                         return false;
205
206
                 }
207
            }
            return true;
209
        }
210
211
        /**
         * Returns all tunnels that are in this board
213
214
         * Oreturn all tunnels that are in this board
215
         */
216
        List<Tunnel> getTunnels();
217
218
        /**
         * Returns all rooms that are in this board
220
221
         * Oreturn all rooms that are in this board
222
         */
223
        List<Room> getRooms();
225
        boolean isGoalRoomLevel();
226
228
         * Returns a {@linkplain MarkSystem mark system} that can be used to avoid checking
229
       twice a tile
         * Oreturn a mark system
231
         * @see MarkSystem
232
         */
233
        MarkSystem getMarkSystem();
234
235
236
         * Returns the {@linkplain MarkSystem mark system} used by the {@link
       #findReachableCases(int)} algorithm
238
         * Oreturn the reachable mark system
239
         * @see MarkSystem
240
        MarkSystem getReachableMarkSystem();
242
243
        // SETTERS //
246
^{247}
        /**
```

```
* Apply the consumer on every tile info
250
        * Oparam consumer the consumer to apply
251
        */
252
       void forEach(Consumer<TileInfo> consumer);
253
254
255
        * Set at tile at the specified index. The index will be converted to
        * cartesian coordinate with {@link #getX(int)} and {@link #getY(int)}
257
258
        * Oparam index index in the board
259
        * Oparam tile the new tile
        * @throws IndexOutOfBoundsException if the index lead to a position outside the board
261
262
       void setAt(int index, Tile tile);
263
265
        * Set at tile at (x, y)
266
267
        * Oparam x x position in the board
        * Oparam y y position in the board
269
        * @throws IndexOutOfBoundsException if the position is outside the board
270
271
       void setAt(int x, int y, Tile tile);
273
274
        * Puts the crates of the given state in the content array.
275
276
        * Oparam state The state with the crates
277
        */
278
       void addStateCrates(State state);
280
281
        * Removes the crates of the given state from the content array.
282
283
        * Oparam state The state with the crates
284
        */
285
       void removeStateCrates(State state);
286
288
        * Puts the crates of the given state in the content array.
289
        * If a crate is outside the board, it doesn't throw an {@link
290
       IndexOutOfBoundsException}
291
        * Oparam state The state with the crates
292
        */
293
       void safeAddStateCrates(State state);
294
295
296
        * Removes the crates of the given state from the content array.
297
        * If a crate is outside the board, it doesn't throw an {@link
298
       IndexOutOfBoundsException}
299
        \ast Oparam state The state with the crates
300
301
       void safeRemoveStateCrates(State state);
302
303
       // -----
       // * Methods used by solvers
305
       // * You need to call #initForSolver() first *
306
       307
308
```

```
* Initialize the board for solving:
310
         * 
311
               compute floor tiles: an array containing all non-wall tile
312
               compute {@linkplain #computeDeadTiles() dead tiles}
313
               find {@linkplain #findTunnels() tunnels}
314
         * 
315
         * <strong>The board must have no crate inside</strong>
317
         * @see Tunnel
318
        */
319
       void initForSolver();
321
322
        * Creates or recreates the floor array. It is an array containing all tile info
323
         * that are not a wall
325
        void computeFloors();
326
327
        /**
         * Apply the consumer on every tile info except walls
329
330
        * Oparam consumer the consumer to apply
331
        */
332
        void forEachNotWall(Consumer<TileInfo> consumer);
333
334
        /**
335
        * Compute which tunnel contains a crate
336
        * Oparam state current state
337
         */
338
        void computeTunnelStatus(State state);
340
341
        * Compute packing order progress for each room if the level
342
        * is a goal room level
343
        * Oparam state current state
344
        */
345
       void computePackingOrderProgress(State state);
346
        // *******
348
        // * ANALYSIS *
349
        // *******
350
       // * STATIC *
352
353
354
        * Detects the dead positions of a level. Dead positions are cases that make the level
355
       unsolvable
        * when a crate is put on them.
356
        * After this function has been called, to check if a given crate at (x,y) is a dead
        * you can use {@link TileInfo#isDeadTile()} to check in constant time.
358
        * The board <strong>MUST</strong> have <strong>NO CRATES</strong> for this function
359
    \hookrightarrow to work.
360
       void computeDeadTiles();
361
362
        * Find tunnels. A tunnel is something like this:
364
         * 
365
               $$$$$$
366
                    $$$$$
```

```
$$$$
                  $$$$$$$
369
         * 
370
        * 
         * A tunnel doesn't contain a target
372
373
        void findTunnels();
374
        /**
376
         * Finds room based on tunnel. Basically all tile that aren't in a tunnel are in room.
377
        * This means that you need to call {@link #findTunnels()} before!
378
        * A room that contains a target is a packing room.
        */
380
        void findRooms();
381
382
        /**
        * Compute packing order. No crate should be on the board
384
385
        void tryComputePackingOrder();
386
        // * DYNAMIC *
388
389
        * Find reachable tiles
392
         * Oparam playerPos The indic of the case on which the player currently is.
393
        void findReachableCases(int playerPos);
395
396
397
        * This method compute the top left reachable position of the player of pushing a
        * at crate to crateDest. It is used to calculate the position
399
        * of the player in a {@link State}.
400
        * This is also an example of use of {@link MarkSystem}
401
        * Oreturn the top left reachable position after pushing the crate
403
        * @see MarkSystem
404
        * @see Mark
406
        int topLeftReachablePosition(TileInfo crate, TileInfo crateDest);
407
        /**
        * Oparam tile tile
410
        * @return the corral in which {@code tile} is
411
         */
412
        Corral getCorral(TileInfo tile);
414
415
        * @return the {@link CorralDetector} used to find corrals
416
417
        CorralDetector getCorralDetector();
418
   }
419
```

#### State

```
package fr.valax.sokoshell.solver;
import fr.valax.sokoshell.utils.SizeOf;
import java.util.Arrays;
import java.util.Random;
```

```
7
    * A state represents an arrangement of the crates in the board and the location of the
   \rightarrow player.
10
    * @implNote <strong>DO NOT MODIFY THE ARRAY AFTER THE INITIALIZATION. THE HASH WON'T BE
11
   → RECALCULATED</strong>
    * @author darth-mole
    * @author PoulpoGaz
13
14
   public class State {
15
       // http://sokobano.de/wiki/index.php?title=Solver#Hash_Function
17
       // https://en.wikipedia.org/wiki/Zobrist_hashing
18
       protected static int[][] zobristValues;
19
21
        * Oparam minSize minSize is the number of tile in the board
22
23
       public static void initZobristValues(int minSize) {
25
           if (zobristValues == null) {
26
                i = 0;
                zobristValues = new int[minSize][2];
           } else if (zobristValues.length < minSize) {</pre>
29
                i = zobristValues.length;
30
                zobristValues = Arrays.copyOf(zobristValues, minSize);
           } else {
32
                i = zobristValues.length;
33
34
           Random random = new Random();
36
           for (; i < zobristValues.length; i++) {</pre>
37
                if (zobristValues[i] == null) {
38
                    zobristValues[i] = new int[2];
               }
40
41
               zobristValues[i][0] = random.nextInt();
42
                zobristValues[i][1] = random.nextInt();
           }
44
       }
45
48
       protected final int playerPos;
49
       protected final int[] cratesIndices;
50
       protected final int hash;
       protected final State parent;
52
53
       public State(int playerPos, int[] cratesIndices, State parent) {
           this(playerPos, cratesIndices, hashCode(playerPos, cratesIndices), parent);
55
56
57
       public State(int playerPos, int[] cratesIndices, int hash, State parent) {
           this.playerPos = playerPos;
           this.cratesIndices = cratesIndices;
60
           this.hash = hash:
61
           this.parent = parent;
       }
63
64
65
        * Creates a child of the state.
```

```
* It uses property of XOR to compute efficiently the hash of the child state
         * Oparam newPlayerPos the new player position
68
         * Oparam crateToMove the index of the crate to move
69
         * Oparam crateDestination the new position of the crate to move
70
         * @return the child state
71
72
       public State child(int newPlayerPos, int crateToMove, int crateDestination) {
73
            int[] newCrates = this.cratesIndices().clone();
            int hash = this.hash ^ zobristValues[this.playerPos][0] ^
                zobristValues[newPlayerPos][0] // 'moves' the player in the hash
                     ^ zobristValues[newCrates[crateToMove]][1] ^
76
                     → zobristValues[crateDestination][1]; // 'moves' the crate in the hash
            newCrates[crateToMove] = crateDestination;
77
            return new State(newPlayerPos, newCrates, hash, this);
       }
81
       public long approxSizeOfAccurate() {
82
            return SizeOf.getStateLayout().instanceSize() +
83
                    SizeOf.getIntArrayLayout().instanceSize() +
                     (long) Integer.BYTES * cratesIndices.length;
85
       }
86
       public long approxSizeOf() {
88
            return 32 +
89
90
                     (long) Integer.BYTES * cratesIndices.length;
91
        }
92
93
94
         * The index of the case of the board on which the player is.
96
       public int playerPos() {
97
            return playerPos;
98
        }
99
100
        /**
101
         * The index of the cases of the board on which the crates are.
102
       public int[] cratesIndices() {
104
            return cratesIndices;
105
       }
106
107
       public int hash() {
108
            return hash;
109
        }
110
111
112
         * The state in which the board was before coming to this state.
113
114
       public State parent() {
115
            return parent;
116
117
119
        @Override
120
        public boolean equals(Object o) {
121
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
123
124
            State state = (State) o;
125
```

```
if (playerPos != state.playerPos) return false;
127
            return equals(cratesIndices, state.cratesIndices);
128
        }
129
130
        /**
131
         * Returns true if all elements of array1 are included in array2 and vice-versa.
132
         * However, because there is no duplicate and the two array have the same length,
133
         * it is only necessary to check if array1 is included in array2.
135
         * Oparam array1 the first array
136
         * Oparam array2 the second array
137
         * Oreturn true if all elements are included in the second one
139
        private boolean equals(int[] array1, int[] array2) {
140
            for (int a : array1) {
141
                 if (!contains(a, array2)) {
                     return false;
143
144
            }
145
            return true;
147
        }
148
        private boolean contains(int a, int[] array) {
150
            for (int b : array) {
151
                 if (a == b) {
152
                     return true;
                 }
154
            }
155
156
            return false;
        }
158
159
        @Override
160
        public int hashCode() {
161
            return hash;
162
163
164
        public static int hashCode(int playerPos, int[] cratesIndices) {
            int hash = zobristValues[playerPos][0];
166
167
            for (int crate : cratesIndices) {
168
                hash ^= zobristValues[crate][1];
            }
170
171
            return hash;
172
        }
173
174
        @Override
175
        public String toString() {
            StringBuilder sb = new StringBuilder();
177
            sb.append("Player: ").append(playerPos).append(", Crates: [");
178
179
            for (int i = 0; i < cratesIndices.length; i++) {</pre>
                 int crate = cratesIndices[i];
                 sb.append(crate);
182
183
                 if (i + 1 < cratesIndices.length) {</pre>
                     sb.append("; ");
185
                 }
186
            }
187
188
```

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

return sb.toString();

}

}
```

# ReachableTiles

```
package fr.valax.sokoshell.solver;
   import fr.valax.sokoshell.solver.board.Board;
3
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.mark.FixedSizeMarkSystem;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   public class ReachableTiles {
       protected final FixedSizeMarkSystem reachable;
10
11
       public ReachableTiles(Board board) {
12
           reachable = new FixedSizeMarkSystem(board.getWidth() * board.getHeight());
13
14
15
       public boolean isReachable(TileInfo tile) {
           return reachable.isMarked(tile.getIndex());
17
18
19
       public void findReachableCases(TileInfo origin) {
20
           reachable.unmarkAll();
           findReachableCases_aux(origin);
22
       }
23
       private void findReachableCases_aux(TileInfo tile) {
25
           reachable.mark(tile.getIndex());
26
           for (Direction d : Direction.VALUES) {
27
               TileInfo adjacent = tile.adjacent(d);
29
               // the second part of the condition avoids to check already processed cases
30
               if (!adjacent.isSolid() && !isReachable(adjacent)) {
                   findReachableCases_aux(adjacent);
               }
33
           }
34
       }
35
   }
```

## Solver

```
package fr.valax.sokoshell.solver;
import java.util.List;

/**
    * Defines the basics for all sokoban solver

* *
    * @author darth-mole
    * @author PoulpoGaz

*/
public interface Solver {

String DFS = "DFS";
String BFS = "BFS";
```

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

### DeadlockTable

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

```
protected static final int NOT A DEADLOCK = 0;
22
       protected static final int MAYBE A DEADLOCK = 1;
23
       protected static final int A_DEADLOCK = 2;
24
       protected static final DeadlockTable DEADLOCK = new DeadlockTable(A_DEADLOCK);
26
       protected static final DeadlockTable NOT_DEADLOCK = new DeadlockTable(NOT_A_DEADLOCK);
27
       protected final int deadlock;
29
30
       protected final int x; // relative to player x
31
       protected final int y; // relative to player y
       protected final DeadlockTable floorChild;
33
       protected final DeadlockTable wallChild;
34
       protected final DeadlockTable crateChild;
       private DeadlockTable(int deadlock) {
37
           this(deadlock, -1, -1, null, null, null);
38
39
       public DeadlockTable(int deadlock, int x, int y,
41
                            DeadlockTable floorChild, DeadlockTable wallChild, DeadlockTable
42
                             this.deadlock = deadlock;
43
           this.x = x;
44
           this.y = y;
45
           this.floorChild = floorChild;
           this.wallChild = wallChild;
47
           this.crateChild = crateChild;
48
       }
49
       public boolean isDeadlock(TileInfo player, Direction pushDir) {
51
           Board board = player.getBoard();
52
53
           if (player.adjacent(pushDir).isCrateOnTarget()) {
               return false;
           }
56
           return switch (pushDir) {
               case LEFT -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.y,
59

→ player.getY() + t.x));
               case UP -> isDeadlock((t) -> board.safeGetAt(player.getX() + t.x,
               → player.getY() + t.y));
               case RIGHT -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.y,
61

→ player.getY() - t.x));
               case DOWN -> isDeadlock((t) -> board.safeGetAt(player.getX() - t.x,
62
                → player.getY() - t.y));
           };
63
64
       private boolean isDeadlock(Function<DeadlockTable, TileInfo> getTile) {
66
           if (deadlock == A_DEADLOCK) {
67
               return true;
           } else if (deadlock == NOT_A_DEADLOCK) {
               return false;
70
71
72
           TileInfo tile = getTile.apply(this);
           if (tile == null) {
75
               return false;
76
           }
```

```
78
            return switch (tile.getTile()) {
79
                case FLOOR -> floorChild.isDeadlock(getTile);
80
                case WALL -> wallChild.isDeadlock(getTile);
                case CRATE -> crateChild.isDeadlock(getTile);
82
                default -> false;
83
            };
84
        }
86
       public static void write(DeadlockTable root, Path out) throws IOException {
87
            try (OutputStream os = new BufferedOutputStream(Files.newOutputStream(out))) {
                Stack<DeadlockTable> stack = new Stack<>();
                stack.push(root);
90
91
                while (!stack.isEmpty()) {
                    DeadlockTable table = stack.pop();
94
                    os.write(table.deadlock);
95
                    if (table.deadlock == MAYBE_A_DEADLOCK) {
96
                         writeInt(os, table.x);
                         writeInt(os, table.y);
98
                         stack.push(table.crateChild);
99
                         stack.push(table.wallChild);
                         stack.push(table.floorChild);
101
                    }
102
                }
103
            }
104
        }
105
106
       public static DeadlockTable read(Path in) throws IOException {
107
            try (InputStream is = new BufferedInputStream(Files.newInputStream(in))) {
                return read(is);
109
110
        }
111
112
       private static DeadlockTable read(InputStream is) throws IOException {
113
            int i = is.read();
114
115
            if (i < 0 | | i > 2) {
                throw new IOException("Malformed table");
117
118
119
            if (i == A_DEADLOCK) {
                return DEADLOCK;
121
            } else if (i == NOT_A_DEADLOCK) {
122
                return NOT_DEADLOCK;
            } else {
124
                int x = readInt(is);
125
                int y = readInt(is);
126
                DeadlockTable floor = read(is);
128
                DeadlockTable wall = read(is);
129
                DeadlockTable crate = read(is);
130
131
                return new DeadlockTable(MAYBE_A_DEADLOCK, x, y, floor, wall, crate);
            }
133
       }
134
       private static void writeInt(OutputStream os, int val) throws IOException {
136
            os.write(val & 0xFF);
137
            os.write((val >> 8) & 0xFF);
138
            os.write((val >> 16) & 0xFF);
139
```

```
os.write((val >> 24) & 0xFF);
140
        }
141
142
       private static int readInt(InputStream is) throws IOException {
143
            int a = is.read() & 0xFF;
144
            int b = is.read() & 0xFF;
145
            int c = is.read() & 0xFF;
146
            int d = is.read() & 0xFF;
148
            return (d << 24) | (c << 16) | (b << 8) | a;
149
        }
150
       public static int countNotDetectedDeadlock(DeadlockTable table, int size) {
152
            Board board = createBoard(size);
153
            // no dead tiles by default
            board.setAt(1, 1, Tile.TARGET);
156
            board.setAt(board.getWidth() - 2, board.getHeight() - 2, Tile.TARGET);
157
158
            board.computeFloors();
            board.computeDeadTiles();
160
            board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
161
162
            return countNotDetectedDeadlock(table, board, board.getWidth() / 2,
163
                board.getHeight() - 3);
        }
164
165
       private static int countNotDetectedDeadlock(DeadlockTable table, Board board, int
166
        → playerX, int playerY) {
            if (table.deadlock == A_DEADLOCK) {
167
                State state = createState(board, playerX, playerY);
169
                // but dead tiles aren't computed...
170
                if (FreezeDeadlockDetector.checkFreezeDeadlock(board, state)) {
171
                    return 0;
172
                }
173
174
                CorralDetector detector = board.getCorralDetector();
175
                detector.findCorral(board, playerX, playerY);
                detector.findPICorral(board, state.cratesIndices());
177
178
                boolean deadlock = false;
179
                for (Corral c : detector.getCorrals()) {
180
                    if (c.isDeadlock(state, true)) {
181
                         deadlock = true;
182
                         break;
183
184
                }
185
186
                if (deadlock) {
                    return 0;
188
                } else {
189
                    BasicStyle.XSB_STYLE.print(board, playerX, playerY);
190
                    return 1; // not detected !
192
                }
193
            } else if (table.deadlock == MAYBE A DEADLOCK) {
194
                int n = countNotDetectedDeadlock(table.floorChild, board, playerX, playerY);
196
                board.setAt(playerX + table.x, playerY + table.y, Tile.WALL);
197
                n += countNotDetectedDeadlock(table.wallChild, board, playerY, playerY);
198
199
```

```
board.setAt(playerX + table.x, playerY + table.y, Tile.CRATE);
200
                n += countNotDetectedDeadlock(table.crateChild, board, playerX, playerY);
201
202
                board.setAt(playerX + table.x, playerY + table.y, Tile.FLOOR);
204
                return n;
205
            } else {
206
                return 0;
            }
208
        }
209
210
        public static DeadlockTable generate(int size) {
212
            // if size = 3, returned board looks like:
213
            // #######
            // #
            // #
                      #
216
            // #
                     #
217
            // # @
                     #
218
            // ######
            // size of generated pattern: size * size
220
            Board board = createBoard(size);
221
            board.setAt(board.getWidth() / 2, board.getHeight() - 4, Tile.CRATE);
224
            return generate(board, createOrder(size), 0, board.getWidth() / 2,
225

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

```
board.setAt(playerX + relativeX, playerY + relativeY, Tile.FLOOR);
258
                 if (wallChild == NOT DEADLOCK && crateChild == NOT DEADLOCK) {
259
                     return NOT_DEADLOCK;
260
                }
261
262
                DeadlockTable floorChild = generate(board, order, index + 1, playerX,
263
                 → playerY);
264
                 return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY, floorChild,
265

→ wallChild, crateChild);
            } else {
                return NOT_DEADLOCK;
267
            }
268
        }
269
        private static Board createBoard(int size) {
271
            Board board = new MutableBoard(size + 4, size + 4);
272
            State.initZobristValues(board.getWidth() * board.getHeight());
273
            for (int x = 0; x < board.getWidth(); x++) {</pre>
275
                 board.setAt(x, 0, Tile.WALL);
276
                 board.setAt(x, board.getHeight() - 1, Tile.WALL);
            }
279
            for (int y = 0; y < board.getHeight(); y++) {</pre>
280
                 board.setAt(0, y, Tile.WALL);
                 board.setAt(board.getWidth() - 1, y, Tile.WALL);
282
            }
283
284
            return board;
        }
286
287
        protected static int[][] createOrder(int size) {
288
            int[][] order = new int[size * size - 2][2];
289
290
            boolean odd = size % 2 == 1;
291
            int i = 0;
292
            int half = size / 2;
            for (int y = 0; y > -size; y--) {
294
                 for (int x = -half; x < half || (x == half && odd); x++) {
295
                     if (x == 0 \&\& (y == 0 || y == -1)) {
296
                         continue;
                     }
298
299
                     order[i] = new int[] {x, y};
300
                     i++;
301
                 }
302
303
304
305
            return order;
306
        }
307
        private static class GenerateDeadlockTableTask extends RecursiveTaskDeadlockTable> {
309
310
            private static final AtomicInteger COUNTER = new AtomicInteger();
311
            private static final int total = 4_782_969;
313
            private final Board board;
314
            private final int[][] order;
315
            private final int index;
```

```
private final int playerX;
            private final int playerY;
318
            private final boolean check;
319
            public GenerateDeadlockTableTask(Board board, int[][] order, int index, int
321
            → playerX, int playerY, boolean check) {
                this.board = board;
322
                this.order = order;
                this.index = index;
324
                this.playerX = playerX;
325
                this.playerY = playerY;
326
                this.check = check;
            }
328
329
            @Override
            protected DeadlockTable compute() {
                int n = COUNTER.incrementAndGet();
332
333
                if (n % 10_000 == 0) {
334
                    System.out.printf("\%.2f\% - %d\%n", 100f * n / total, n);
                }
336
337
                if (check && isDeadlock_(board, playerX, playerY)) {
                    return DEADLOCK;
                } else if (index < order.length) {</pre>
340
                    int relativeX = order[index][0];
341
                    int relativeY = order[index][1];
342
343
                    GenerateDeadlockTableTask wall = subTask(index, Tile.WALL, true);
344
                    GenerateDeadlockTableTask crate = subTask(index, Tile.CRATE, true);
345
                    wall.fork();
347
                    crate.fork();
348
349
                    DeadlockTable wallChild = wall.join();
350
                    DeadlockTable crateChild = crate.join();
351
352
                    if (wallChild == NOT_DEADLOCK && crateChild == NOT_DEADLOCK) {
353
                         return NOT_DEADLOCK;
                    }
355
356
                    GenerateDeadlockTableTask floor = subTask(index, Tile.FLOOR, false);
357
                    DeadlockTable floorChild = floor.fork().join();
359
                    // the three are never equals to deadlock because
360
                    // it means the current board is a deadlock, and
361
                    // it must be detected by isDeadlock_
                    return new DeadlockTable(MAYBE_A_DEADLOCK, relativeX, relativeY,
363

→ floorChild, wallChild, crateChild);
364
                } else {
365
                    return NOT_DEADLOCK;
366
                }
367
            }
            private GenerateDeadlockTableTask subTask(int index, Tile replacement, boolean
370
            MutableBoard board = new MutableBoard(this.board);
                int relativeX = order[index][0];
372
                int relativeY = order[index][1];
373
374
                board.setAt(playerX + relativeX, playerY + relativeY, replacement);
```

```
return new GenerateDeadlockTableTask(board, order, index + 1, playerX,
377
                 → playerY, check);
            }
378
        }
379
380
381
383
384
        private static boolean isDeadlock_(Board board, int playerX, int playerY) {
385
            State first = createState(board, playerX, playerY);
387
            ReachableTiles reachableTiles = new ReachableTiles(board);
388
            HashSet<State> visited = new HashSet<>();
            Queue<State> toVisit = new ArrayDeque<>();
391
            visited.add(first);
392
            toVisit.offer(first);
393
            boolean deadlock = true;
395
            while (!toVisit.isEmpty() && deadlock) {
396
                State parent = toVisit.poll();
398
                board.addStateCrates(parent);
399
400
                if (FreezeDeadlockDetector.checkFreezeDeadlock(board, parent)) {
401
                     board.removeStateCrates(parent);
402
                     continue;
403
                }
404
                reachableTiles.findReachableCases(board.getAt(parent.playerPos()));
406
                deadlock = addChildrenStates(reachableTiles, parent, board, visited, toVisit);
407
                board.removeStateCrates(parent);
408
            }
409
410
            board.addStateCrates(first);
411
412
            return deadlock;
        }
414
415
        private static boolean addChildrenStates (ReachableTiles reachableTiles, State parent,
416
                                                    Board board, Set<State> visited, Queue<State>
                                                     → toVisit) {
            for (int i = 0; i < parent.cratesIndices().length; i++) {</pre>
418
                TileInfo crate = board.getAt(parent.cratesIndices()[i]);
419
                for (Direction dir : Direction.VALUES) {
421
                     TileInfo player = crate.adjacent(dir.negate());
422
423
                     if (!reachableTiles.isReachable(player)) {
424
                         continue;
425
                     }
426
427
                     TileInfo dest = crate.adjacent(dir);
                     if (dest.isSolid()) {
429
                         continue:
430
                     }
431
432
                     State child;
433
                     if (dest.getY() == 1 || dest.getX() == 1 || dest.getX() ==
434
                        board.getWidth() - 2) {
```

```
// remove the crate, it is outside the pattern
                          if (parent.cratesIndices().length == 1) {
436
                              return false; // all crates were moved outside the pattern. not a
437

→ deadlock...

                          }
438
439
                          int topLeft = board.topLeftReachablePosition(crate, board.getAt(0,
440
                          \hookrightarrow 0));
441
                          child = new State(topLeft,
442

→ copyRemoveOneElement(parent.cratesIndices(), i), parent);
                     } else {
444
                          int topLeft = board.topLeftReachablePosition(crate, dest);
445
                          child = parent.child(topLeft, i, dest.getIndex());
446
447
448
                     if (visited.add(child)) {
449
                          toVisit.add(child);
450
                     }
                 }
452
            }
453
            return true; // not a deadlock
        }
456
457
        private static int[] copyRemoveOneElement(int[] array, int indexToRemove) {
458
            int[] newArray = new int[array.length - 1];
459
460
            int offset = 0;
461
            for (int i = 0; i < array.length; i++) {</pre>
                 if (indexToRemove == i) {
463
                     offset = 1;
464
                 } else {
465
                     newArray[i - offset] = array[i];
466
                 }
467
            }
468
469
            return newArray;
        }
471
472
        private static State createState(Board board, int playerX, int playerY) {
473
            List<Integer> ints = new ArrayList<>();
475
            board.forEach(t -> {
476
                 if (t.anyCrate()) {
477
                     ints.add(t.getIndex());
479
            });
480
481
            return new State(playerY * board.getWidth() + playerX, ints.stream().mapToInt(i ->
482

    i).toArray(), null);

        }
483
   }
```

## AStarSolver

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

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

```
import fr.valax.sokoshell.commands.AbstractCommand;
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import fr.valax.sokoshell.solver.collections.SolverPriorityQueue;
   import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
   import fr.valax.sokoshell.solver.heuristic.Heuristic;
11
   import fr.valax.sokoshell.solver.heuristic.SimpleHeuristic;
   import org.jline.reader.Candidate;
   import org.jline.reader.LineReader;
14
15
   import java.io.IOException;
16
   import java.util.List;
18
   public class AStarSolver extends AbstractSolver<WeightedState> {
19
20
       private Heuristic heuristic;
21
       private int lowerBound;
22
23
       public AStarSolver() {
24
           super(A_STAR);
26
27
       @Override
       protected void init(SolverParameters parameters) {
           String heuristicName = parameters.getArgument("heuristic");
30
31
           if (heuristicName.equalsIgnoreCase("simple")) {
32
               heuristic = new SimpleHeuristic(board);
33
34
               heuristic = new GreedyHeuristic(board);
35
           }
37
           toProcess = new SolverPriorityQueue();
38
       }
39
40
       @Override
       protected void addInitialState(Level level) {
42
           final State s = level.getInitialState();
43
           lowerBound = heuristic.compute(s);
45
           toProcess.addState(new WeightedState(s, 0, lowerBound));
46
       }
47
       @Override
49
       protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
50
           if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
               return;
           }
53
54
           final int i = board.topLeftReachablePosition(crate, crateDest);
           // The new player position is the crate position
56
           WeightedState s = toProcess.cachedState().child(i, crate.getCrateIndex(),
57

    crateDest.getIndex());
           s.setHeuristic(heuristic.compute(s));
           if (processed.add(s)) {
60
               toProcess.addState(s);
61
           }
       }
63
64
       Olverride
65
       protected void addParameters(List<SolverParameter> parameters) {
```

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

## Corral

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

```
import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.Tunnel;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.*;
   public class Corral {
10
       public static final int POTENTIAL_PI_CORRAL = 0;
12
       public static final int IS_A_PI_CORRAL = 1;
13
       public static final int NOT_A_PI_CORRAL = 2;
14
       protected final int id;
16
       protected final Board board;
17
       protected int topX;
       protected int topY;
20
21
       protected final Set<Corral> adjacentCorrals = new HashSet<>();
22
24
        * All crates that are inside the corral and surrounding the corral
25
       protected final List<TileInfo> barrier = new ArrayList<>();
       protected final List<TileInfo> crates = new ArrayList<>();
28
       protected boolean containsPlayer;
29
       protected boolean adjacentToPlayerCorral; // the player corral is adjacent to itself
       protected int isPICorral;
31
       protected boolean onlyCrateOnTarget; // true if all crates in crates list are crate on
32

→ target

       protected boolean isValid = false;
34
35
       protected final Set<CorralState> visited = new HashSet<>();
36
       protected final Queue<CorralState> toVisit = new ArrayDeque<>();
37
       protected final ReachableTiles reachable;
       protected CorralState currentState;
39
       protected DeadlockTable deadlockTable;
40
       public Corral(int id, Board board) {
42
           this.id = id;
43
           this.board = board;
           this.reachable = new ReachableTiles(board);
       }
46
47
       public boolean isDeadlock(State originalState) {
           return isDeadlock(originalState, false);
50
51
       public boolean isDeadlock(State originalState, boolean forceContainsAllCrate) {
           if (!isPICorral() ||
53
                   onlyCrateOnTarget ||
54
                   !forceContainsAllCrate && crates.size() ==
                    → originalState.cratesIndices().length) {
               return false;
56
           }
57
           addFrozenCrates(originalState);
           if (!forceContainsAllCrate && crates.size() ==
           → originalState.cratesIndices().length) {
               return false;
61
           }
```

```
63
            boolean deadlock = true;
64
            CorralState firstState = removeOutsideCrate(originalState);
65
            visited.add(firstState);
67
            toVisit.add(firstState);
68
69
            while (!toVisit.isEmpty() && deadlock) {
                currentState = toVisit.remove();
71
72
                board.addStateCrates(currentState);
73
                if (FreezeDeadlockDetector.checkFreezeDeadlock(board, currentState)) {
75
                    board.removeStateCrates(currentState);
76
                    continue;
                }
79
                board.computeTunnelStatus(currentState);
80
                reachable.findReachableCases(board.getAt(currentState.playerPos()));
81
                deadlock = addChildrenStates();
83
                board.removeStateCrates(currentState);
84
                if (visited.size() >= 1000) {
86
                    deadlock = false;
87
                }
88
            }
            visited.clear();
91
            toVisit.clear();
92
            // re-add crates
94
            board.addStateCrates(originalState);
95
96
97
            return deadlock;
        }
98
99
        private void addFrozenCrates(State state) {
100
            for (int i : state.cratesIndices) {
                TileInfo crate = board.getAt(i);
102
103
                if (crates.contains(crate)) {
104
                    continue;
                }
106
107
                if (isFrozen(crate, Direction.LEFT) && isFrozen(crate, Direction.UP)) {
108
                    crates.add(crate);
109
                }
110
            }
111
        }
112
113
114
         * True if the crate is almost frozen ie right now it can be moved
115
         * in the axis: it happens when an adjacent tile on the axis is solid.
         * The adjacent tile must be in the corral is it is a crate
118
        private boolean isFrozen(TileInfo tile, Direction axis) {
119
            TileInfo left = tile.adjacent(axis);
            TileInfo right = tile.adjacent(axis.negate());
121
122
            return left.isWall() ||
123
                    left.anyCrate() && crates.contains(left) ||
```

```
right.isWall() ||
125
                     right.anyCrate() && crates.contains(right);
126
        }
127
128
129
130
         * Oreturn false if not a deadlock
131
         */
        private boolean addChildrenStates() {
133
            int[] cratesIndices = currentState.cratesIndices();
134
135
            boolean deadlock = true;
            for (int i = 0; i < cratesIndices.length && deadlock; i++) {
137
                TileInfo crate = board.getAt(cratesIndices[i]);
138
139
                if (crate.isInATunnel()) {
                     deadlock = addChildrenStatesInTunnel(i, crate);
141
                } else {
142
                     deadlock = addChildrenStatesDefault(i, crate);
143
                }
            }
145
146
            return deadlock;
147
        }
149
150
        // THE TWO FOLLOWING METHODS ARE COPIED FROM ABSTRACT SOLVER.
        // I hope that one day, I will change that
152
        //
153
154
        protected boolean addChildrenStatesInTunnel(int crateIndex, TileInfo crate) {
            // the crate is in a tunnel. two possibilities: move to tunnel.startOut or
156

→ tunnel.endOut

            // this part of the code assume that there is no other crate in the tunnel.
157
            // normally, this is impossible...
158
159
            for (Direction pushDir : Direction.VALUES) {
160
                TileInfo player = crate.adjacent(pushDir.negate());
161
                 if (reachable.isReachable(player)) {
163
                     TileInfo dest = crate.getTunnelExit().getExit(pushDir);
164
165
                     if (dest != null && !dest.isSolid()) {
                         if (!addState(crateIndex, crate, dest, pushDir)) {
167
                             return false; // not a deadlock
168
                         }
169
                     }
                }
171
            }
172
            return true;
174
        }
175
176
        protected boolean addChildrenStatesDefault(int crateIndex, TileInfo crate) {
177
            for (Direction d : Direction.VALUES) {
                TileInfo crateDest = crate.adjacent(d);
179
                if (crateDest.isSolid()) {
180
                     continue; // The destination case is not empty
                }
182
183
                if (crateDest.isDeadTile()) {
184
                     continue; // Useless to push a crate on a dead position
185
```

```
}
186
187
                TileInfo player = crate.adjacent(d.negate());
188
                if (!reachable.isReachable(player)) {
                     // The player cannot reach the case to push the crate
190
                     // also checks if tile is solid: a solid tile is never reachable
191
                     continue;
192
                }
194
195
                // check for tunnel
196
                Tunnel tunnel = crateDest.getTunnel();
198
                // the crate will be pushed inside the tunnel
199
                if (tunnel != null) {
200
                     if (tunnel.crateInside()) { // pushing inside will lead to a corral
                         deadlock
                         continue;
202
203
                     // ie the crate can't be pushed to the other extremities of the tunnel
205
                     // however, sometimes (boxxle 24) it is useful to push the crate inside
206
                     // the tunnel. That's why the second addState is done (after this if)
207
                     // and only if this tunnel isn't oneway
208
                     if (!tunnel.isPlayerOnlyTunnel()) {
209
                         TileInfo newDest = null;
210
                         Direction pushDir = null;
212
                         if (crate == tunnel.getStartOut()) {
213
                             if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate())
214
                                 {
                                  newDest = tunnel.getEndOut();
215
                                  pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
216
                             }
217
                         } else {
218
                             if (tunnel.getStartOut() != null &&
219
                                 !tunnel.getStartOut().anyCrate()) {
                                  newDest = tunnel.getStartOut();
220
                                  pushDir = tunnel.getStart().direction(tunnel.getStartOut());
                             }
222
                         }
223
                         if (newDest != null && !newDest.isDeadTile()) {
                             if (!addState(crateIndex, crate, newDest, pushDir)) {
226
                                  return false;
227
                             }
                         }
                     }
230
231
                     if (tunnel.isOneway()) {
                         continue;
233
                     }
234
                }
235
                if (!addState(crateIndex, crate, crateDest, d)) {
                     return false;
238
                }
239
            }
241
            return true;
242
        }
243
244
```

```
/**
         * Oreturn false if not a deadlock
246
247
        private boolean addState(int crateIndex, TileInfo crate, TileInfo dest, Direction
            pushDir) {
            // a crate can be moved outside the corral
249
            if (!isInCorral(dest)) {
250
                return false;
            }
252
253
            if (deadlockTable.isDeadlock(crate.adjacent(pushDir.negate()), pushDir)) {
254
                return true; // current state is a deadlock, we need to continue the research
256
257
            // all crates of the corral can be moved to a target
            int n = 0;
260
            for (int i : currentState.cratesIndices()) {
261
                 if (i != crate.getIndex() && board.getAt(i).isCrateOnTarget()) {
262
                }
264
            }
265
            if (dest.isTarget() && n + 1 == currentState.cratesIndices.length) { // TODO:
267
                crate may be on target
                return false;
268
            }
269
            // create sub state
271
            int newPlayerPos = board.topLeftReachablePosition(crate, dest);
272
            CorralState sub = currentState.child(newPlayerPos, crateIndex, dest.getIndex());
274
            if (crate.isCrate() && dest.isTarget()) {
275
                 sub.increaseNumberOnTarget();
276
            } else if (crate.isCrateOnTarget() && dest.isFloor()) {
277
                 sub.decreaseNumberOnTarget();
            }
279
280
            if (visited.add(sub)) {
                toVisit.offer(sub);
282
283
284
            return true;
        }
286
287
         * Remove crates that are not part of the corral
289
         * and create a new state without these crates
290
         * Oparam state current state
291
         * @return a state without crate outside the corral
         */
293
        private CorralState removeOutsideCrate(State state) {
294
            int numOnTarget = 0;
295
            int[] newCrates = new int[crates.size()];
            int[] oldCrates = state.cratesIndices();
298
            int j = 0;
299
            for (int i = 0; i < oldCrates.length; i++) {</pre>
                TileInfo crate = board.getAt(oldCrates[i]);
301
                if (isInCorral(oldCrates[i])) {
302
303
                     if (crate.isCrateOnTarget()) {
304
                         numOnTarget++;
```

```
}
306
                     newCrates[j] = oldCrates[i];
307
                     j++;
                 } else {
309
                     crate.removeCrate();
310
                 }
311
            }
313
            CorralState corralState = new CorralState(state.playerPos(), newCrates, null);
314
            corralState.setNumOnTarget(numOnTarget);
315
            return corralState;
        }
317
318
        private boolean isInCorral(int crate) {
319
            TileInfo tile = board.getAt(crate);
321
            return crates.contains(tile);
322
        }
323
324
        private boolean isInCorral(TileInfo tile) {
325
            Corral c = board.getCorral(tile);
326
            if (c == null) {
                 return isInCorral(tile.getIndex());
329
            } else {
330
                 return c == this;
            }
332
        }
333
334
        public int getTopX() {
            return topX;
336
337
338
        public int getTopY() {
            return topY;
340
341
342
        public List<TileInfo> getBarrier() {
            return barrier;
344
345
346
        public List<TileInfo> getCrates() {
            return crates;
348
349
350
        public boolean containsPlayer() {
351
            return containsPlayer;
352
353
354
        public boolean isPICorral() {
355
            return isPICorral == IS_A_PI_CORRAL;
356
357
        public DeadlockTable getDeadlockTable() {
            return deadlockTable;
360
        }
361
        public void setDeadlockTable(DeadlockTable deadlockTable) {
363
            this.deadlockTable = deadlockTable;
364
        }
365
```

```
@Override
367
        public int hashCode() {
368
            return id;
369
370
371
        @Override
372
        public boolean equals(Object o) {
373
            if (this == o) return true;
            if (!(o instanceof Corral corral)) return false;
375
376
            return id == corral.id;
377
        }
379
        private static class CorralState extends State {
380
            private int numOnTarget;
383
            public CorralState(int playerPos, int[] cratesIndices, State parent) {
384
                 super(playerPos, cratesIndices, parent);
385
            }
387
            public CorralState(int playerPos, int[] cratesIndices, int hash, State parent) {
388
                 super(playerPos, cratesIndices, hash, parent);
390
391
            private CorralState(State state) {
392
                 super(state.playerPos, state.cratesIndices, state.hash, state.parent);
394
395
            @Override
396
            public CorralState child(int newPlayerPos, int crateToMove, int crateDestination)
                 return new CorralState(super.child(newPlayerPos, crateToMove,
398

    crateDestination));
            }
399
400
            public void increaseNumberOnTarget() {
401
                 numOnTarget++;
402
404
            public void decreaseNumberOnTarget() {
405
                 numOnTarget--;
406
            }
408
            public int getNumOnTarget() {
409
                 return numOnTarget;
410
            }
411
412
            public void setNumOnTarget(int numOnTarget) {
413
                 this.numOnTarget = numOnTarget;
414
            }
415
        }
416
    }
417
```

## BruteforceSolver

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

import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverCollection;
```

```
import java.util.ArrayDeque;
9
   * This class serves as a base class for DFS and BFS solvers, as these class are nearly
10
   * difference being in the order in which they treat the states (LIFO for DFS and FIFO for
11
   → BFS).
12
   public abstract class BruteforceSolver extends AbstractSolver<State> {
13
14
       public BruteforceSolver(String name) {
15
           super(name);
17
18
       public static DFSSolver newDFSSolver() {
19
           return new DFSSolver();
21
22
       public static BFSSolver newBFSSolver() {
23
           return new BFSSolver();
25
26
       @Override
27
       protected void addInitialState(Level level) {
           toProcess.addState(level.getInitialState());
29
30
31
       @Override
32
       protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
33
           if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
34
               return;
           }
36
37
           final int i = board.topLeftReachablePosition(crate, crateDest);
38
           // The new player position is the crate position
           State s = toProcess.cachedState().child(i, crate.getCrateIndex(),

    crateDest.getIndex());
41
           if (processed.add(s)) {
               toProcess.addState(s);
43
           }
44
       }
45
       @Override
47
       public int lowerBound() {
48
           return -1;
49
       }
51
52
        * Base class for DFS and BFS solvers collection (both of them use {@link
      ArrayDeque}), the only difference being in
        * which side of the queue is used (end => FIFO => DFS, start => LIFO => BFS)
54
55
       \verb|private| static| abstract| class| BasicBruteforceSolverCollection| implements|
       → SolverCollection<State> {
57
           protected final ArrayDeque<State> collection = new ArrayDeque<>();
58
           protected State cachedState;
60
61
           @Override
62
           public void clear() {
```

```
collection.clear();
            }
65
66
            @Override
67
            public boolean isEmpty() {
68
                 return collection.isEmpty();
69
70
            @Override
72
            public int size() {
73
                return collection.size();
74
            }
76
            @Override
77
            public void addState(State state) {
                 collection.offer(state);
80
81
            @Override
82
            public State peekAndCacheState() {
                 cachedState = popState();
84
                return cachedState;
85
            }
            @Override
88
            public State cachedState() {
89
                return cachedState;
            }
91
        }
92
93
        private static class DFSSolver extends BruteforceSolver {
95
            public DFSSolver() {
96
                 super(DFS);
97
            }
98
            @Override
100
            protected void init(SolverParameters parameters) {
101
                 toProcess = new DFSSolverCollection();
103
104
            private static class DFSSolverCollection extends BasicBruteforceSolverCollection {
105
                 @Override
107
                public State popState() {
108
                     return collection.removeLast();
109
110
111
                 @Override
112
                 public State peekState() {
113
                     return collection.peekLast();
114
115
            }
116
        }
117
        private static class BFSSolver extends BruteforceSolver {
119
120
            public BFSSolver() {
                 super(BFS);
123
124
            @Override
```

```
protected void init(SolverParameters parameters) {
                 toProcess = new BFSSolverCollection();
127
128
129
            private static class BFSSolverCollection extends BasicBruteforceSolverCollection {
130
131
                 @Override
132
                 public State popState() {
                     return collection.removeFirst();
134
135
136
                 @Override
                 public State peekState() {
138
                     return collection.peekFirst();
139
140
141
            }
142
        }
143
   }
144
```

#### Tracker

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

#### AbstractSolver

```
package fr.valax.sokoshell.solver;
2
   import fr.valax.sokoshell.graphics.style.BasicStyle;
3
   import fr.valax.sokoshell.solver.board.*;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import fr.valax.sokoshell.solver.collections.SolverCollection;
   import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
   import fr.valax.sokoshell.utils.SizeOf;
   import java.io.IOException;
10
   import java.nio.file.Path;
   import java.util.*;
13
14
   * This class is the base for bruteforce-based solvers, i.e. solvers that use an
   \rightarrow exhaustive search to try and find a
   * solution.
   * @author darth-mole
17
   */
   public abstract class AbstractSolver<S extends State> implements Trackable, Solver {
19
20
       protected static final String TIMEOUT = "timeout";
21
       protected static final String MAX_RAM = "max-ram";
       protected static final String ACCURATE = "accurate";
24
       protected final String name;
25
       protected final DeadlockTable table;
28
       protected SolverCollection<S> toProcess;
29
       protected final Set<State> processed = new HashSet<>();
31
       protected MutableBoard board;
32
33
       private boolean running = false;
       private boolean stopped = false;
35
36
       // statistics
37
       private long timeStart = -1;
       private long timeEnd = -1;
39
       private int nStateProcessed = -1;
40
       private int queueSize = -1;
       private Tracker tracker;
43
       public AbstractSolver(String name) {
44
           this.name = name;
           try {
47
               table = DeadlockTable.read(Path.of("4x4.table"));
           } catch (IOException e) {
               throw new RuntimeException(e);
50
           }
51
       }
52
       @Override
54
       public SolverReport solve(SolverParameters params) {
55
           Objects.requireNonNull(params);
           // init statistics, timeout and stop
58
           String endStatus = null;
59
```

```
60
            running = true;
61
            stopped = false;
62
            long timeout = params.getArgument(TIMEOUT);
64
            long maxRam = params.getArgument(MAX_RAM);
65
            boolean accurate = params.getArgument(ACCURATE);
66
            if (accurate) {
68
                SizeOf.initialize();
69
            }
70
            timeStart = System.currentTimeMillis();
72
            timeEnd = -1;
73
            nStateProcessed = 0;
            queueSize = 0;
76
            if (tracker != null) {
77
                tracker.reset();
78
            }
80
            // init the research
81
            Level level = params.getLevel();
83
84
            State.initZobristValues(level.getWidth() * level.getHeight());
85
            final State initialState = level.getInitialState();
87
            State finalState = null;
88
89
            board = new MutableBoard(level);
            board.removeStateCrates(initialState);
91
            board.initForSolver();
92
            board.getCorralDetector().setDeadlockTable(table);
93
94
            init(params);
95
            processed.clear();
96
97
            addInitialState(level);
99
            if (level.getPack().name().equals("XSokoban_90") && level.getIndex() == 3) {
100
                board.getAt(9, 10).setDeadTile(true);
101
            }
102
103
            while (!toProcess.isEmpty() && !stopped) {
104
                if (hasTimedOut(timeout)) {
105
                     endStatus = SolverReport.TIMEOUT;
106
                     break;
107
                }
108
109
                if (hasRamExceeded(maxRam, accurate)) {
110
                     endStatus = SolverReport.RAM_EXCEED;
111
                     break;
112
                }
113
                S state = toProcess.peekAndCacheState();
115
                board.addStateCrates(state);
116
117
                if (board.isCompletedWith(state)) {
118
                     finalState = state;
119
                     break;
120
                }
121
```

```
122
                int playerX = board.getX(state.playerPos());
123
                int playerY = board.getY(state.playerPos());
124
125
                CorralDetector detector = board.getCorralDetector();
126
                detector.findCorral(board, playerX, playerY);
127
128
                if (checkPICorralDeadlock(state)) {
                    board.removeStateCrates(state);
130
                     continue;
131
                }
132
                // compute after checking for corral deadlock, as corral deadlock deals with
134

    tunnels

                board.computeTunnelStatus(state);
135
                board.computePackingOrderProgress(state);
137
                addChildrenStates(board.getAt(playerX, playerY));
138
                board.removeStateCrates(state);
139
            }
141
            // END OF RESEARCH
142
            timeEnd = System.currentTimeMillis();
            nStateProcessed = processed.size();
145
            queueSize = toProcess.size();
146
147
            // 'free' ram
148
            processed.clear();
149
            toProcess.clear();
150
            board = null;
152
            running = false;
153
154
            System.out.println("END: " + finalState + " - " + endStatus);
155
156
            if (endStatus != null) {
157
                return SolverReport.withoutSolution(params, getStatistics(), endStatus);
            } else if (stopped) {
                return SolverReport.withoutSolution(params, getStatistics(),
160

→ SolverReport.STOPPED);
            } else if (finalState != null) {
161
                return SolverReport.withSolution(finalState, params, getStatistics());
            } else {
163
                return SolverReport.withoutSolution(params, getStatistics(),
164

    SolverReport.NO_SOLUTION);

            }
165
        }
166
167
168
         * Initialize the solver. This method is called after the initialization of
169
170
         */
171
       protected abstract void init(SolverParameters parameters);
172
       protected abstract void addInitialState(Level level);
174
175
        protected boolean checkPICorralDeadlock(State state) {
            CorralDetector detector = board.getCorralDetector();
177
            detector.findPICorral(board, state.cratesIndices());
178
179
            for (Corral corral : detector.getCorrals()) {
```

```
if (corral.isDeadlock(state)) {
181
                     return true;
182
                 }
183
            }
185
            return false;
186
        }
187
        protected void addChildrenStates(TileInfo player) {
189
            Corral playerCorral = board.getCorralDetector().findCorral(player);
190
191
            List<TileInfo> crates = playerCorral.getCrates();
            for (int i = 0; i < crates.size(); i++) {</pre>
193
                 TileInfo crateTile = crates.get(i);
194
                 // check if the crate is already at his destination
                 if (board.isGoalRoomLevel() && crateTile.isInARoom()) {
197
                     Room r = crateTile.getRoom();
198
199
                     if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
201
                     } else {
202
                         tryGoalCut(crateTile);
                     }
204
                 }
205
206
                 Tunnel tunnel = crateTile.getTunnel();
207
                 if (tunnel != null) {
208
                     addChildrenStatesInTunnel(crateTile);
209
                 } else {
210
                     addChildrenStatesDefault(crateTile);
                 }
212
            }
213
        }
214
215
        protected void tryGoalCut(TileInfo crate) {
216
            TileInfo player = board.getAt(currentState().playerPos());
217
218
            // only works because rooms have one entry
            CrateAStar crateAStar = board.getCrateAStar();
220
            List<Room> rooms = board.getRooms();
221
            for (int i = 0; i < rooms.size(); i++) {</pre>
222
                 Room r = rooms.get(i);
224
                 Tunnel tunnel = r.getTunnels().get(0);
225
                 TileInfo entrance;
                 if (tunnel.getStartOut().getRoom() == r) {
227
                     entrance = tunnel.getStartOut();
228
                 } else {
229
                     entrance = tunnel.getEndOut();
                 }
231
232
                 if (r.isGoalRoom() && r.getPackingOrderIndex() >= 0) {
233
                     if (crateAStar.hasPath(player, null, crate, entrance)) {
                         addStateCheckForGoalMacro(crate, entrance, null);
236
                 }
237
            }
        }
239
240
241
        protected void addChildrenStatesInTunnel(TileInfo crate) {
```

```
// the crate is in a tunnel. two possibilities: move to tunnel.startOut or
            // this part of the code assume that there is no other crate in the tunnel.
243
            // normally, this is impossible...
244
245
            for (Direction pushDir : Direction.VALUES) {
246
                TileInfo player = crate.adjacent(pushDir.negate());
247
                if (player.isReachable()) {
249
                    TileInfo dest = crate.getTunnelExit().getExit(pushDir);
250
251
                     if (dest != null && !dest.isSolid()) {
                         addStateCheckForGoalMacro(crate, dest, pushDir);
253
254
                }
            }
        }
257
258
        protected void addChildrenStatesDefault(TileInfo crate) {
259
            for (Direction d : Direction.VALUES) {
261
                TileInfo crateDest = crate.adjacent(d);
262
                if (crateDest.isSolid()) {
                     continue; // The destination case is not empty
264
265
266
                if (crateDest.isDeadTile()) {
267
                     continue; // Useless to push a crate on a dead position
268
                }
269
270
                TileInfo player = crate.adjacent(d.negate());
                if (!player.isReachable()) {
272
                     // The player cannot reach the case to push the crate
273
                    // also checks if tile is solid: a solid tile is never reachable
274
                    continue;
275
                }
277
278
                // check for tunnel
                Tunnel tunnel = crateDest.getTunnel();
280
281
                // the crate will be pushed inside the tunnel
282
                if (tunnel != null) {
                    if (tunnel.crateInside()) { // pushing inside will lead to a corral
284
                        deadlock
                         continue;
285
                    }
286
287
                    // ie the crate can't be pushed to the other extremities of the tunnel
288
                    // however, sometimes (boxxle 24) it is useful to push the crate inside
                    // the tunnel. That's why the second addState is done (after this if)
290
                    // and only if this tunnel isn't oneway
291
                    if (!tunnel.isPlayerOnlyTunnel()) {
292
                         TileInfo newDest = null;
293
                         Direction pushDir = null;
295
                         if (crate == tunnel.getStartOut()) {
296
                             if (tunnel.getEndOut() != null && !tunnel.getEndOut().anyCrate())
                                 newDest = tunnel.getEndOut();
298
                                 pushDir = tunnel.getEnd().direction(tunnel.getEndOut());
299
                             }
```

```
} else {
                                                                 if (tunnel.getStartOut() != null &&
302
                                                                         !tunnel.getStartOut().anyCrate()) {
                                                                          newDest = tunnel.getStartOut();
303
                                                                          pushDir = tunnel.getStart().direction(tunnel.getStartOut());
304
                                                                }
305
                                                       }
306
                                                       if (newDest != null && !newDest.isDeadTile()) {
308
                                                                 addStateCheckForGoalMacro(crate, newDest, pushDir);
309
                                                       }
310
                                             }
312
                                              if (tunnel.isOneway()) {
313
                                                       continue;
314
                                             }
                                    }
316
317
                                    addStateCheckForGoalMacro(crate, crateDest, d);
318
                           }
                  }
320
321
                 \verb|protected| void| \verb|addStateCheckForGoalMacro| (TileInfo| crate, TileInfo| dest, Direction| | tileInfo| crate, TileInfo| dest, Direction| | tileInfo| dest, Di
                         pushDir) {
                           Room room = dest.getRoom();
323
                           if (room != null && board.isGoalRoomLevel() && room.getPackingOrderIndex() >= 0) {
324
                                    // goal macro!
                                    TileInfo newDest = room.getPackingOrder().get(room.getPackingOrderIndex());
326
327
                                    addState(crate, newDest, null);
328
                           } else {
                                    addState(crate, dest, pushDir);
330
331
                  }
332
333
334
                    * Check if the move leads to a deadlock.
335
                    * Only for simple deadlock that don't require
336
                    * lots of computation like PI Corral deadlock
338
                    * Oparam crate crate to move
339
                    * Oparam crateDest crate destination
                    * Oparam pushDir push dir of the player. If the move is a macro move,
                                                           it is the last push done by the player. It can be null
342
                    * @return true if deadlock
343
                    */
344
                  protected boolean checkDeadlockBeforeAdding(TileInfo crate, TileInfo crateDest,
345
                   → Direction pushDir) {
                           crate.removeCrate();
346
                           crateDest.addCrate();
347
348
                           boolean deadlock = FreezeDeadlockDetector.checkFreezeDeadlock(crateDest);
349
350
                           if (!deadlock && pushDir != null) {
                                    deadlock = table.isDeadlock(crateDest.adjacent(pushDir.negate()), pushDir);
353
354
                           crate.addCrate();
                           crateDest.removeCrate();
356
357
358
                           return deadlock;
                 }
```

```
360
        /**
361
         * Add a state to the processed set. If it wasn't already added, it is added to
362
         * the toProcess queue. The move is unchecked
364
         * Oparam crate crate to move
365
         * Oparam crateDest crate destination
366
         * @param pushDir push dir of the player. If the move is a macro move,
                            it is the last push done by the player. It can be null
368
369
        protected abstract void addState(TileInfo crate, TileInfo crateDest, Direction
370
        → pushDir);
371
        protected boolean hasTimedOut(long timeout) {
372
            return timeout > 0 && timeout + timeStart < System.currentTimeMillis();</pre>
375
        protected boolean hasRamExceeded(long maxRam, boolean accurate) {
376
            if (\max Ram > 0) {
377
                 State curr = currentState();
379
                 if (curr != null) {
380
                     long stateSize;
                     long ramUsed;
                     if (accurate) {
383
                         stateSize = curr.approxSizeOfAccurate();
384
                         ramUsed = SizeOf.approxSizeOfAccurate(processed, stateSize);
                     } else {
386
                         stateSize = curr.approxSizeOf();
387
                         ramUsed = SizeOf.approxSizeOf(processed, stateSize);
388
                     }
390
                     return ramUsed + toProcess.size() * stateSize >= maxRam;
391
                 }
392
            }
393
394
            return false;
395
        }
396
        @Override
398
        public String getName() {
399
            return name;
400
        }
402
        @Override
403
        public boolean isRunning() {
404
            return running;
405
406
407
        @Override
408
        public boolean stop() {
409
            stopped = true;
410
            return true;
411
        }
412
414
        @Override
415
        public List<SolverParameter> getParameters() {
            List<SolverParameter> params = new ArrayList<>();
417
            addParameters(params);
418
            return params;
419
        }
```

```
421
        /**
422
         * Add your parameters to the list returned by {@link #getParameters()}
423
         * Oparam parameters parameters that will be returned by {Olink #getParameters()}
424
         */
425
        protected void addParameters(List<SolverParameter> parameters) {
426
            parameters.add(new SolverParameter.Long(TIMEOUT, "Maximal runtime of the solver",
427

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

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

### SolverParameter

```
package fr.valax.sokoshell.solver;
2
   import fr.poulpogaz.json.IJsonReader;
   import fr.poulpogaz.json.IJsonWriter;
   import fr.poulpogaz.json.JsonException;
   import fr.valax.sokoshell.commands.AbstractCommand;
   import org.jline.reader.Candidate;
   import org.jline.reader.LineReader;
   import java.io.IOException;
10
   import java.util.List;
   import java.util.Objects;
12
   import java.util.regex.Matcher;
   import java.util.regex.Pattern;
14
15
16
    * A parameter given to a {@link Solver}. A parameter has a name and a description.
17
    * It is responsible for parsing arguments and give default value. Implementations
    * can also define how to auto complete and must implements \{0 \mid ink \mid from Json(IJsonReader)\}
    * and {@link #toJson(IJsonWriter)}
20
21
   public abstract class SolverParameter {
22
       protected final String name;
24
       protected final String description;
25
       public SolverParameter(String name, String description) {
           this.name = name;
28
           this.description = description;
29
       }
31
       public String getName() {
32
           return name;
33
35
```

```
public String getDescription() {
36
           return description;
37
38
       public abstract void set(String argument) throws AbstractCommand.InvalidArgument;
40
41
       public abstract Object get();
42
       public Object getOrDefault() {
44
           Object o = get();
45
46
           if (o == null) {
47
                o = Objects.requireNonNull(getDefaultValue());
48
49
           return o;
       }
52
53
       public abstract Object getDefaultValue();
54
       public boolean hasArgument() {
56
           return get() != null;
57
       }
60
       public void complete(LineReader reader, String argument, List<Candidate> candidates) {
61
62
       }
63
64
65
        * @implNote name is already writen
67
       public abstract void toJson(IJsonWriter jw) throws JsonException, IOException;
68
69
       /**
70
        * @implNote name is already read
71
72
       public abstract void from Json (IJson Reader jr) throws Json Exception, IO Exception;
73
75
76
77
       public static class Integer extends SolverParameter {
79
80
           protected final int defaultValue;
           protected java.lang.Integer value = null;
82
83
           public Integer(String name, int defaultValue) {
84
                this(name, null, defaultValue);
           }
86
87
           public Integer(String name, String description, int defaultValue) {
                super(name, description);
                this.defaultValue = defaultValue;
90
           }
91
92
           @Override
           public void set(String argument) throws AbstractCommand.InvalidArgument {
94
                try {
95
                    value = java.lang.Integer.parseInt(argument);
96
                } catch (NumberFormatException e) {
```

```
throw new AbstractCommand.InvalidArgument(e);
98
                 }
99
            }
100
101
            @Override
102
            public Object get() {
103
                 return value;
104
106
            @Override
107
            public Object getDefaultValue() {
108
                 return defaultValue;
110
111
            @Override
112
            public void toJson(IJsonWriter jw) throws JsonException, IOException {
                 if (value != null) {
114
                     jw.value(value);
115
                 }
116
            }
118
            @Override
119
            public void fromJson(IJsonReader jr) throws JsonException, IOException {
                 value = jr.nextInt();
122
        }
123
        public static class Long extends SolverParameter {
125
126
            protected final long defaultValue;
127
            protected java.lang.Long value = null;
129
            public Long(String name, long defaultValue) {
130
                 this(name, null, defaultValue);
131
            }
132
133
            public Long(String name, String description, long defaultValue) {
134
                 super(name, description);
135
                 this.defaultValue = defaultValue;
            }
137
138
            @Override
139
            public void set(String argument) throws AbstractCommand.InvalidArgument {
                 try {
141
                     value = java.lang.Long.parseLong(argument);
142
                 } catch (NumberFormatException e) {
143
                     throw new AbstractCommand.InvalidArgument(e);
                 }
145
            }
146
147
            @Override
148
            public Object get() {
149
                 return value;
150
151
            @Override
153
            public Object getDefaultValue() {
154
                 return defaultValue;
            }
156
157
            Olverride
158
            public void toJson(IJsonWriter jw) throws JsonException, IOException {
```

```
if (value != null) {
160
                     jw.value(value);
161
162
            }
163
164
            @Override
165
            public void fromJson(IJsonReader jr) throws JsonException, IOException {
166
                 value = jr.nextLong();
168
        }
169
170
        public static class Boolean extends SolverParameter {
172
173
            protected final boolean defaultValue;
            protected java.lang.Boolean value = null;
176
            public Boolean(String name, boolean defaultValue) {
177
                 this(name, null, defaultValue);
178
            }
180
            public Boolean(String name, String description, boolean defaultValue) {
181
                 super(name, description);
                 this.defaultValue = defaultValue;
            }
184
185
            @Override
            public void set(String argument) throws AbstractCommand.InvalidArgument {
187
                 try {
188
                     int v = java.lang.Integer.parseInt(argument);
189
                     value = v != 0;
191
                 } catch (NumberFormatException e) {
192
                     value = java.lang.Boolean.parseBoolean(argument);
193
                 }
194
            }
195
196
            @Override
197
            public Object get() {
                 return value;
199
200
201
            @Override
            public Object getDefaultValue() {
203
                 return defaultValue;
204
            }
205
206
            @Override
207
            public void toJson(IJsonWriter jw) throws JsonException, IOException {
208
                 if (value != null) {
                     jw.value(value);
210
                 }
211
            }
212
213
            public void fromJson(IJsonReader jr) throws JsonException, IOException {
215
                 value = jr.nextBoolean();
216
            }
        }
218
219
220
```

```
public static class RamParameter extends Long {
223
            private static final Pattern PATTERN = Pattern.compile("^(\\d+)\\s*([gmk])?b$",
224
            → Pattern.CASE_INSENSITIVE);
225
            public RamParameter(String name, long defaultValue) {
226
                super(name, "Maximal ram usage of the solver", defaultValue);
227
229
            public RamParameter(String name, String description, long defaultValue) {
230
                super(name, description, defaultValue);
231
233
            @Override
234
            public void set(String argument) throws AbstractCommand.InvalidArgument {
                Matcher matcher = PATTERN.matcher(argument);
237
                if (matcher.matches() && matcher.groupCount() >= 1 && matcher.groupCount() <=</pre>
238
                    long r = java.lang.Long.parseLong(matcher.group(1));
240
                    if (matcher.groupCount() == 2) {
241
                         String unit = matcher.group(2).toLowerCase();
242
                         r = switch (unit) {
244
                             case "g" -> r * 1024 * 1024 * 1024;
245
                             case "m" -> r * 1024 * 1024;
                             case "k" -> r * 1024;
247
                             default -> throw new
248
                              → AbstractCommand.InvalidArgument("Invalid ram argument");
                         };
                    }
250
251
                    value = r;
252
                } else {
253
                    throw new AbstractCommand.InvalidArgument("Invalid ram argument");
                }
255
            }
256
        }
257
   }
258
```

## Trackable

```
package fr.valax.sokoshell.solver;
  import fr.valax.sokoshell.solver.board.Board;
  /**
5
   * A solver that implements this interface allows
    * other objects to get information about the current
    * research.
    * <br>
    * Methods are by default non-synchronized and <strong>should not</strong>
   * modify the state of the solver.
    * Implementations are free to violate the first term of the contract
12
    * <strong>(not the second)</strong>, but they must indicate it.
13
  public interface Trackable extends Solver {
15
16
17
        * Oreturn the number of state explored or -1
        */
19
```

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

#### CorralDetector

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

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

import java.util.*;

/**

* A union find structure to find corral in a map.
```

```
* The objective of this object is to compute corral,
    * barriers and topY, topX position of each corral.
12
    */
13
   @SuppressWarnings("ForLoopReplaceableByForEach")
14
   public class CorralDetector {
15
16
       private final Corral[] corrals;
17
       private final int[] parent;
       private final int[] rank;
19
20
       private final Set<Corral> currentCorrals;
21
       private int realNumberOfCorral;
23
24
       public CorralDetector(Board board) {
25
           int size = board.getWidth() * board.getHeight();
           parent = new int[size];
27
           rank = new int[size];
28
29
           corrals = new Corral[size];
           for (int i = 0; i < parent.length; i++) {</pre>
31
                parent[i] = i;
32
                corrals[i] = new Corral(i, board);
           }
35
           currentCorrals = new HashSet<>(size);
36
       }
37
38
39
        * Find corral. Compute topX, topY. Find the corral that
40
        * contains the player.
41
        * Other values (isPICorral, crates, barriers) are not
42
        * valid after a call to this method. Use {@link #findPICorral(Board, int[])}
43
        * to revalidate them.
44
45
        * @param board the board
46
        * @param playerX player position x
47
        * Oparam playerY player position y
48
        */
       public void findCorral(Board board, int playerX, int playerY) {
50
           currentCorrals.clear();
51
52
           int h = board.getHeight();
           int w = board.getWidth();
54
55
           for (int y = 1; y < h - 1; y++) {
56
                TileInfo left = board.getAt(0, y);
58
                for (int x = 1; x < w - 1; x++) {
59
                    TileInfo t = board.getAt(x, y);
61
                    if (!t.isSolid()) {
62
                        TileInfo up = board.getAt(x, y - 1);
63
                        if (!up.isSolid() && !left.isSolid()) {
65
                            addToCorral(t, up);
66
                            mergeTwoCorrals(up, left);
67
                        } else if (!up.isSolid()) {
                            addToCorral(t, up);
69
                        } else if (!left.isSolid()) {
70
                            addToCorral(t, left);
71
                        } else {
```

```
newCorral(t);
73
                          }
74
                     } else {
75
                         int i = t.getIndex();
76
                         parent[i] = -1;
77
                         rank[i] = -1;
78
                         corrals[i].isValid = false;
79
                     }
81
                     left = t;
82
                 }
83
            }
85
            int playerCorral = find(playerY * board.getWidth() + playerX);
86
            corrals[playerCorral].containsPlayer = true;
            realNumberOfCorral = currentCorrals.size();
89
        }
90
91
        /**
         * Find PI corral
93
         * Oparam board the board
94
         * Oparam crates crates on the board
95
96
        public void findPICorral(Board board, int[] crates) {
97
            preComputePICorral(board, crates);
98
99
            List<Corral> corrals = new ArrayList<>(currentCorrals);
100
101
            for (int i = 0; i < corrals.size(); i++) {</pre>
102
                 Corral c = corrals.get(i);
104
                 if (!c.containsPlayer()) {
105
                     if (isPICorral(c)) {
106
                         c.isPICorral = Corral.IS_A_PI_CORRAL;
107
                         corrals.remove(i);
108
                         i--;
109
                     }
110
                 } else {
                     c.isPICorral = Corral.NOT_A_PI_CORRAL;
112
                     corrals.remove(i);
113
                     i--;
114
                 }
            }
116
117
            for (Corral c : corrals) {
118
                 if (c.isValid && c.isPICorral == Corral.POTENTIAL_PI_CORRAL) {
119
                     mergeWithAdjacents(board, c);
120
121
            }
        }
123
124
        protected boolean isICorral(Corral corral) {
125
            for (TileInfo crate : corral.barrier) {
126
                 for (Direction dir : Direction.VALUES) {
                     TileInfo crateDest = crate.adjacent(dir);
128
                     if (crateDest.isSolid()) {
129
                         continue;
                     }
131
132
                     TileInfo player = crate.adjacent(dir.negate());
133
                     if (player.isSolid()) {
134
```

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

```
if (adj.isPICorral()) {
196
                         return;
197
198
                     if (!adj.containsPlayer) {
200
                         break;
201
                     }
202
                 }
204
                 corral = fullyMergeTwoCorrals(board, corral, adj);
205
            }
206
            if (isPICorral(corral)) {
208
                 corral.isPICorral = Corral.IS_A_PI_CORRAL;
209
                 corral.isPICorral = Corral.NOT_A_PI_CORRAL;
            }
212
        }
213
214
        private Corral fullyMergeTwoCorrals(Board board, Corral a, Corral b) {
            Corral corral = mergeTwoCorrals(board.getAt(a.getTopX(), a.getTopY()),
216
             → board.getAt(b.getTopX(), b.getTopY()));
217
            if (corral == b) {
                 b = a; // this way, we can deal with corral (before a) and b, without doing
219
                 \hookrightarrow disjonction.
            }
220
221
            // Merge properties. It is assumed that a and b doesn't contain the player
222
            // topX, topY are already updated
223
            // the set currentCorrals was also updated.
            corral.adjacentToPlayerCorral |= b.adjacentToPlayerCorral;
225
            corral.onlyCrateOnTarget &= b.onlyCrateOnTarget;
226
227
            // update adjacentCorrals
228
            // Add all adjacents corral of b to corral, but corral is adjacent to b,
            // we must remove it. The remove is done before addAll because the resulting
230
            // set is likely to be bigger than b one.
231
            b.adjacentCorrals.remove(corral);
            // also update adjacent of b
233
            for (Corral bAdj : b.adjacentCorrals) {
234
                bAdj.adjacentCorrals.remove(b);
235
                 if (bAdj != corral) {
237
                     bAdj.adjacentCorrals.add(corral);
238
                 }
239
            }
            corral.adjacentCorrals.remove(b);
241
            corral.adjacentCorrals.addAll(b.adjacentCorrals);
242
243
            // update barrier and crates
244
            for (TileInfo tile : b.crates) {
245
                 if (!corral.crates.contains(tile)) {
246
                     corral.crates.add(tile);
247
                 }
            }
249
250
            // merge the two barrier. Some crates aren't in a barrier.
            for (TileInfo tile : b.barrier) {
252
                 if (!corral.barrier.contains(tile)) {
253
254
                     corral.barrier.add(tile);
                 }
```

```
}
257
258
            int[] adjacents = new int[4];
259
            int size;
260
            for (int i = 0; i < corral.barrier.size(); i++) {</pre>
261
                 TileInfo crate = corral.barrier.get(i);
262
                 size = 0;
                 for (Direction dir : Direction.VALUES) {
264
                     TileInfo tile = crate.adjacent(dir);
265
                     if (tile.isSolid()) {
266
                          continue;
                     }
268
269
                     Corral adj = findCorral(tile);
270
                     boolean new_ = true;
272
                     for (int k = 0; k < size; k++) {
273
                          if (adjacents[k] == adj.id) {
274
                              new_ = false;
                              break;
276
                          }
277
                     }
278
                     if (new_) {
280
                          adjacents[size] = adj.id;
281
                          size++;
282
                     }
283
                 }
284
285
                 if (size <= 1) { // not in barrier !</pre>
                     corral.barrier.remove(i);
287
                     i--;
288
                 }
289
            }
290
291
            return corral;
292
        }
293
295
         * Compute adjacent corrals of crates, barriers and various property of Corral
296
297
        protected void preComputePICorral(Board board, int[] crates) {
            List<Corral> adj = new ArrayList<>();
299
300
            for (int crateI : crates) {
301
                 TileInfo crate = board.getAt(crateI);
302
303
                 adj.clear();
304
                 // find adjacent corrals
306
                 boolean adjacentToPlayerCorral = false;
307
                 for (Direction dir : Direction.VALUES) {
308
                     TileInfo tile = crate.adjacent(dir);
309
                     if (tile.isSolid()) {
                          continue;
311
                     }
312
                     Corral corral = findCorral(tile);
314
                     // maximal size of adj is 4, so I think that using a list rather than a
315

→ set is faster

                     if (!adj.contains(corral)) {
```

```
adj.add(corral);
                     }
318
319
                     if (corral.containsPlayer()) {
                          adjacentToPlayerCorral = true;
321
322
                 }
323
                 if (adj.size() == 1) {
325
                     // the crate is inside a corral
326
                     // and not a part of a barrier
327
                     adj.get(0).crates.add(crate);
329
                     if (crate.isCrate()) {
330
                         adj.get(0).onlyCrateOnTarget = false;
331
                 } else if (adj.size() > 1) {
333
                     // crate is a part of a barrier
334
                     for (int i = 0; i < adj.size(); i++) {</pre>
335
                         Corral corral = adj.get(i);
                         corral.crates.add(crate);
337
                         corral.barrier.add(crate);
338
                         corral.adjacentToPlayerCorral |= adjacentToPlayerCorral;
340
                          if (crate.isCrate()) {
341
                              corral.onlyCrateOnTarget = false;
342
                         }
343
344
                         for (int j = i + 1; j < adj.size(); j++) {
345
                              Corral corral2 = adj.get(j);
346
                              if (corral.adjacentCorrals.add(corral2)) {
348
                                  corral2.adjacentCorrals.add(corral);
349
350
                         }
351
                     }
352
                 }
353
            }
354
        }
356
357
         * Move a node from a aPackage to another. {@code node}
358
         * and {@code dest} must be in separate trees.
         * This method breaks the union find structure.
360
         * So, it must be used carefully.
361
         */
362
        private void addToCorral(TileInfo tile, TileInfo inCorral) {
363
            int i = tile.getIndex();
364
            int rootI = find(inCorral.getIndex());
365
            parent[i] = rootI;
367
            rank[i] = 0;
368
            rank[rootI] = Math.max(1, rank[rootI]);
369
        }
370
372
         * Remove a node from his aPackage and create a new aPackage.
373
         * This method breaks the union find structure.
         * So, it must be used carefully.
375
376
        private void newCorral(TileInfo tile) {
377
378
            int i = tile.getIndex();
```

```
parent[i] = i;
            rank[i] = 0;
380
381
            Corral corral = corrals[i];
            corral.containsPlayer = false;
383
            corral.isPICorral = Corral.POTENTIAL_PI_CORRAL;
384
            corral.onlyCrateOnTarget = true;
385
            corral.isValid = true;
            corral.crates.clear();
387
            corral.barrier.clear();
388
            corral.adjacentCorrals.clear();
389
            corral.topX = tile.getX();
            corral.topY = tile.getY();
391
392
            currentCorrals.add(corral);
        }
395
        private Corral mergeTwoCorrals(TileInfo inCorral1, TileInfo inCorral2) {
396
            int corral1I = find(inCorral1.getIndex());
397
            int corral2I = find(inCorral2.getIndex());
399
            if (corral1I != corral2I) {
400
                 int oldCorralI;
                 int newCorralI;
402
                 if (rank[corral1I] < rank[corral2I]) {</pre>
403
                     oldCorralI = corral1I;
404
                     newCorralI = corral2I;
                 } else if (rank[corral1I] > rank[corral2I]) {
406
                     oldCorralI = corral2I;
407
                     newCorralI = corral1I;
408
                 } else {
                     oldCorralI = corral1I;
410
                     newCorralI = corral2I;
411
                     rank[newCorralI]++;
412
                 }
413
                parent[oldCorralI] = newCorralI;
415
416
                 Corral newCorral = corrals[newCorralI];
                 Corral oldCorral = corrals[oldCorralI];
418
419
                 oldCorral.isValid = false;
420
                 currentCorrals.remove(oldCorral);
                 newCorral.containsPlayer |= oldCorral.containsPlayer();
422
423
                 if (oldCorral.topY < newCorral.topY || (oldCorral.topY == newCorral.topY &&</pre>
                    oldCorral.topX < newCorral.topX)) {</pre>
                     newCorral.topX = oldCorral.topX;
425
                     newCorral.topY = oldCorral.topY;
426
                 }
428
                 return newCorral;
429
            }
430
            return corrals[corral1I];
        }
433
434
        private int find(int i) {
436
            if (parent[i] != i) {
437
                 int root = find(parent[i]);
438
                 parent[i] = root;
```

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

# WeightedState

```
package fr.valax.sokoshell.solver;
   import fr.valax.sokoshell.utils.SizeOf;
   /**
5
   * A simple derivation of State with a weight, i.e. something to rank the states.
    * Used for instance by {@link AStarSolver}
   public class WeightedState extends State {
9
       private int cost = 0;
11
12
       private int heuristic = 0;
13
14
       public WeightedState(int playerPos, int[] cratesIndices, int hash, State parent, int
15
           cost, int heuristic) {
           super(playerPos, cratesIndices, hash, parent);
16
           this.setCost(cost);
           this.setHeuristic(heuristic);
18
       }
19
20
       public WeightedState(State state, int cost, int heuristic) {
21
```

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

# Level

```
package fr.valax.sokoshell.solver;
import fr.poulpogaz.json.JsonException;
```

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

```
}
66
67
68
         * Returns the player position on the x-axis at the beginning
70
         * Oreturn the player position on the x-axis at the beginning
71
         */
72
        public int getPlayerX() {
            return playerPos % getWidth();
74
75
76
         * Returns the player position on the y-axis at the beginning
78
79
         * Oreturn the player position on the y-axis at the beginning
80
         */
        public int getPlayerY() {
82
            return playerPos / getWidth();
83
        }
84
        /**
86
         * Returns the initial state i.e. a state representing the level at the beginning
87
         * Oreturn the initial state
89
90
        public State getInitialState() {
91
            State.initZobristValues(getWidth() * getHeight()); // TODO
92
93
            List<Integer> cratesIndices = new ArrayList<>();
94
95
            for (int y = 0; y < getHeight(); y++) {
                for (int x = 0; x < getWidth(); x++) {
97
                     if (getAt(x, y).anyCrate()) {
98
                         cratesIndices.add(y * getWidth() + x);
99
                     }
100
                }
101
            }
102
103
            int[] cratesIndicesArray = new int[cratesIndices.size()];
            for (int i = 0; i < cratesIndices.size(); i++) {</pre>
105
                cratesIndicesArray[i] = cratesIndices.get(i);
106
            }
107
108
            return new State(playerPos, cratesIndicesArray, null);
109
        }
110
111
        public BigInteger estimateNumberOfState() {
112
            if (maxNumberOfStateEstimation == null) {
113
                 // + 1 for numberOfCrate because we also consider the player
114
                maxNumberOfStateEstimation = Utils.binomial(numberOfNonWalls, numberOfCrates +
                 \rightarrow 1);
            }
116
117
            return maxNumberOfStateEstimation;
        }
119
120
        public BigInteger estimateNumberOfState(int nDeadTile) {
121
            int nFloor = numberOfNonWalls - nDeadTile;
123
            return Utils.binomial(nFloor, numberOfCrates + 1);
124
        }
125
126
```

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

```
* Returns all solver reports
189
190
         * Oreturn all solver reports
191
         */
192
        public List<SolverReport> getSolverReports() {
193
            return solverReports;
194
195
        /**
197
         * Returns the number of solver report
198
199
         * Oreturn the number of solver report
         */
201
        public int numberOfSolverReport() {
202
            return solverReports.size();
        }
205
206
         * Add a solver report to this level
207
         * @param solverReport the report to add
209
         * Othrows IllegalArgumentException if the report isn't for this level
210
211
        public synchronized void addSolverReport(SolverReport solverReport) {
            if (solverReport.getParameters().getLevel() != this) {
213
                 throw new
214

→ IllegalArgumentException("Attempting to add a report to the wrong level");

            }
215
            solverReports.add(solverReport);
216
        }
217
        public synchronized void removeSolverReport(int index) {
219
            solverReports.remove(index);
220
221
222
        public synchronized int indexOf(SolverReport solverReport) {
223
            if (solverReport.getParameters().getLevel() != this) {
224
                 return -1;
225
            }
            return solverReports.indexOf(solverReport);
227
        }
228
229
         * Returns if an attempt to solve this level was done. It doesn't mean that this level
231
       has a solution
232
         * @return {@code true} if an attempt to solve this level was done.
233
234
        public boolean hasReport() {
235
            return solverReports.size() > 0;
236
        }
237
238
239
         * Returns {@code true} if this level has a solution
240
         * @return {@code true} if this level has a solution
242
243
        public boolean hasSolution() {
            for (int i = 0; i < solverReports.size(); i++) {</pre>
245
                 SolverReport r = solverReports.get(i);
246
                 if (r.isSolved()) {
247
248
                     return true;
```

```
}
249
            }
250
251
            return false;
252
        }
253
254
255
         * Returns the index of this level in the pack
257
         * @return the index of this level in the pack
258
259
        public int getIndex() {
            return index;
261
262
263
        /**
         * Returns the pack in which this level is
265
266
         \ast Oreturn the pack in which this level is
267
         */
        public Pack getPack() {
269
            return pack;
270
        }
271
273
274
         * A builder of {@link Level}
275
         */
276
        public static class Builder {
277
278
            private int playerX = -1;
            private int playerY = -1;
280
281
            private Tile[][] board = new Tile[0][0];
282
            private int width;
283
            private int height;
284
            private int index;
285
286
            /**
             * Builds and returns a {@link Level}
288
289
             * @return the new {@link Level}
290
              * Othrows BuilderException if the player is outside the board
              * r the player is on a solid tile
292
293
            public Level build() {
294
                 if (board == null) {
295
                     throw new BuilderException("Board is null");
296
297
298
                 if (playerX < 0 || playerX >= width) {
299
                     throw new BuilderException("Player x out of bounds");
300
                 }
301
302
                 if (playerY < 0 || playerY >= height) {
303
                     throw new BuilderException("Player y out of bounds");
304
                 }
305
                 if (board[playerY][playerX].isSolid()) {
307
                     throw new BuilderException("Player is on a solid tile");
308
                 }
309
310
```

```
formatLevel();
312
                 return new Level(board, width, height, playerY * width + playerX, index);
313
            }
314
315
            /**
316
             * Format the level for the solver. Some levels aren't surrounded by wall
317
             * or have rooms that are inaccessible. This method removes these rooms
              * and add wall if necessary.
319
320
            private void formatLevel() {
321
                 Set<Integer> visited = new HashSet<>();
323
                 int i = 0;
324
                 for (int y = 0; y < height; y++) {
                     for (int x = 0; x < width; x++) {
                          if (board[y][x] != Tile.WALL && !visited.contains(i)) {
327
                              addWallIfNecessary(x, y, visited);
328
                          }
329
                          i++;
331
                     }
332
                 }
                 surroundByWallIfNecessary();
335
336
337
            private void addWallIfNecessary(int x, int y, Set<Integer> visited) {
338
                 boolean needWall = true;
339
340
                 Set<Integer> localVisited = new HashSet<>();
                 Stack<Integer> toVisit = new Stack<>();
342
                 toVisit.add(y * width + x);
343
                 localVisited.add(toVisit.peek());
344
345
                 while (!toVisit.isEmpty()) {
346
                     int i = toVisit.pop();
347
348
                     int x2 = i % width;
                     int y2 = i / width;
350
351
                     if (x2 == playerX && y2 == playerY) {
352
                         needWall = false;
354
355
                     for (Direction d : Direction.VALUES) {
356
                          int x3 = x2 + d.dirX();
357
                         int y3 = y2 + d.dirY();
358
359
                          if (x3 < 0 \mid | x3 >= width \mid | y3 < 0 \mid | y3 >= height) {
                              continue;
361
362
363
                          int i3 = y3 * width + x3;
365
                          if (board[y3][x3] != Tile.WALL && localVisited.add(i3)) {
366
                              visited.add(i3);
367
                              toVisit.push(i3);
                         }
369
                     }
370
                 }
371
```

```
if (needWall) {
                     for (Integer i : localVisited) {
374
                          int x2 = i % width;
375
                          int y2 = i / width;
376
377
                          board[y2][x2] = Tile.WALL;
378
                     }
379
                 }
            }
381
382
             private void surroundByWallIfNecessary() {
383
                 int left = 0;
                 int right = 0;
385
                 int top = 0;
386
                 int bottom = 0;
                 for (int y = 0; y < height; y++) {
389
                      if (board[y][0] != Tile.WALL) {
390
                          left = 1;
391
                     }
                     if (board[y][width - 1] != Tile.WALL) {
393
                          right = 1;
394
                     }
                 }
396
397
                 for (int x = 0; x < width; x++) {
398
                     if (board[0][x] != Tile.WALL) {
                          top = 1;
400
401
                     if (board[height - 1][x] != Tile.WALL) {
402
                          bottom = 1;
404
                 }
405
406
                 if (left == 0 && right == 0 && top == 0 && bottom == 0) {
407
                     return;
408
                 }
409
410
                 Tile[][] newTiles = new Tile[height + top + bottom][width + right + left];
412
                 for (int y = 0; y < height + top + bottom; <math>y++) {
413
                     for (int x = 0; x < width + right + left; <math>x++) {
                          if (x \ge left \&\& y \ge left \&\& x < width + left \&\& y < height + top) {
                              newTiles[y][x] = board[y - top][x - left];
416
417
                              newTiles[y][x] = Tile.WALL;
418
                          }
419
                     }
420
                 }
421
                 board = newTiles;
423
                 width += right + left;
424
                 height += top + bottom;
425
                 playerX += left;
426
                 playerY += top;
            }
428
429
             /**
              * Returns the player position on the x-axis
431
432
              * Oreturn the player position on the x-axis
433
434
```

```
public int getPlayerX() {
                 return playerX;
436
437
438
439
             * Returns the player position on the y-axis
440
441
             * Oreturn the player position on the y-axis
442
443
            public int getPlayerY() {
444
                return playerY;
445
            }
447
448
             \ast Set the player position to (x, y)
449
450
             * Oparam x player position on the x-axis
451
             * Oparam y player position on the y-axis
452
453
            public void setPlayerPos(int x, int y) {
                this.playerX = x;
455
                 this.playerY = y;
456
            }
457
459
             * Set the player position on the x-axis to x
460
             * Cparam playerX the new player position on the x-axis
462
463
            public void setPlayerX(int playerX) {
464
                 this.playerX = playerX;
            }
466
467
468
             * Set the player position on the y-axis to x
470
             * Oparam playerY the new player position on the y-axis
471
472
            public void setPlayerY(int playerY) {
                 this.playerY = playerY;
474
475
476
            private void resizeIfNeeded(int minWidth, int minHeight) {
                 setSize(Math.max(minWidth, width),
478
                         Math.max(minHeight, height));
479
            }
480
481
482
             * Resize this level to (newWidth, newHeight). If dimensions are higher than the
483
        old one,
             * new tiles are filled with WALL. For other, tiles are the same.
484
485
             * @param newWidth the new width of the level
486
             * Oparam newHeight the new width of the level
487
488
            public void setSize(int newWidth, int newHeight) {
489
                 if (newWidth == width && newHeight == height) {
490
                     return;
                 }
492
493
                Tile[][] newBoard = new Tile[newHeight][newWidth];
494
```

```
int yMax = Math.min(newHeight, height);
496
                 int xMax = Math.min(newWidth, width);
497
                 for (int y = 0; y < yMax; y++) {
498
                     System.arraycopy(board[y], 0, newBoard[y], 0, xMax);
499
500
                     for (int x = xMax; x < newWidth; x++) {</pre>
501
                         newBoard[y][x] = Tile.WALL;
502
                     }
                 }
504
505
                 board = newBoard;
506
                 width = newWidth;
508
                 height = newHeight;
509
            }
510
512
              * Returns the width of the level
513
514
             * @return the width of the level
516
            public int getWidth() {
517
                 return width;
520
521
             * Sets the width of the level
523
              * Oparam width the new width of the level
524
              * @see #setSize(int, int)
525
             */
            public void setWidth(int width) {
527
                 setSize(width, height);
528
            }
529
530
531
              * Returns the height of the level
532
533
              * Oreturn the height of the level
535
            public int getHeight() {
536
                 return height;
537
            }
539
540
             * Sets the height of the level
541
              * Oparam height the new height of the level
543
              * @see #setSize(int, int)
544
             */
            public void setHeight(int height) {
546
                 setSize(width, height);
547
548
549
             * Set at (x, y) the tile. If (x, y) is outside the level, the level is resized
551
552
             * Oparam tile the new tile
              * Oparam x x position
554
              * @param y y position
555
              */
556
            public void set(Tile tile, int x, int y) {
```

```
resizeIfNeeded(x, y);
                  board[y][x] = tile;
559
             }
560
561
             /**
562
              * Returns the tile at (x, y)
563
              * Oparam x x position of the tile
564
              * Oparam y y position of the tile
              * Oreturn the tile at (x, y)
566
567
             public Tile get(int x, int y) {
568
                  if (x < 0 \mid \mid x >= width \mid \mid y < 0 \mid \mid y >= height) {
                      return null;
570
571
                  return board[y][x];
             }
574
575
576
              * Returns the index of the level
              * @return the index of the level
578
579
             public int getIndex() {
                 return index;
581
582
583
             /**
              * Sets the index of the level
585
              * Oparam index the new index of the level
586
587
             public void setIndex(int index) {
                  this.index = index;
589
590
        }
591
    }
```

## SolverReport

```
package fr.valax.sokoshell.solver;
2
   import fr.poulpogaz.json.JsonException;
   import fr.poulpogaz.json.JsonPrettyWriter;
   import fr.poulpogaz.json.JsonReader;
   import fr.valax.sokoshell.SokoShell;
   import fr.valax.sokoshell.graphics.style.BoardStyle;
   import fr.valax.sokoshell.solver.board.Board;
   import fr.valax.sokoshell.solver.board.Move;
   import fr.valax.sokoshell.solver.board.MutableBoard;
10
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import fr.valax.sokoshell.solver.pathfinder.CrateAStar;
12
   import fr.valax.sokoshell.solver.pathfinder.Node;
13
14
   import java.io.*;
   import java.util.*;
16
   import java.util.stream.Collectors;
17
18
   * An object representing the output of a solver. It contains the parameters given to the
20
   \rightarrow solver,
    * some statistics, the solver status and if the status is {@link
      SolverReport#SOLUTION_FOUND},
```

```
* it contains two representation of the solution: a sequence of {@link State} and a
   → sequence of {@link Move}.
23
   * @see SolverParameters
24
   * @see ISolverStatistics
   * @see State
26
   * @see Move
27
   * @author PoulpoGaz
   * @author darth-mole
30
   public class SolverReport {
31
       public static final String NO_SOLUTION = "No solution";
33
       public static final String SOLUTION_FOUND = "Solution found";
34
       public static final String STOPPED = "Stopped";
35
       public static final String TIMEOUT = "Timeout";
       public static final String RAM_EXCEED = "Ram exceed";
37
38
       * Creates and returns a report that doesn't contain a solution
41
       st Oparam params the parameters of the solver
42
       * Oparam stats the statistics
       * Oparam status the solver status
       * @return a report without a solution
45
       * @throws IllegalArgumentException if the state is {@link

→ SolverReport#SOLUTION_FOUND}

        */
47
       public static SolverReport withoutSolution(SolverParameters params, ISolverStatistics
48
       \hookrightarrow stats, String status) {
           return new SolverReport(params, stats, null, status);
       }
50
51
       * Creates and returns a report containing a solution. The solution is determined
       * from the final state.
55
       * Oparam finalState the final state
56
        * Oparam params the parameters of the solver
        * Oparam stats the statistics
58
        * @return a report with a solution
59
       public static SolverReport withSolution(State finalState, SolverParameters params,
       List<State> solution = new ArrayList<>();
62
63
           State s = finalState;
           while (s.parent() != null)
65
           {
66
               solution.add(s);
               s = s.parent();
68
           }
69
           solution.add(s);
70
           Collections.reverse(solution);
71
           return new SolverReport(params, stats, solution, SOLUTION_FOUND);
73
       }
74
       private final SolverParameters parameters;
       private final ISolverStatistics statistics;
77
78
       private final String status;
```

```
80
        /**
81
         * Solution packed in an int array.
82
         * Three bits are used for storing a move.
         * Move 1 is located at bit 0 of array 0,
         * Move 2 is located at bit 3 of array 0,
85
         * ...,
86
         * Move 10 is located at bit 27 of array 0,
         * Move 11 is located at bit 30 of array 0
88
         * and use the first bit of array 1.
89
         * Move 12 is located at bit 1 of array 1,
90
         * etc.
         * Bits are stored in little-endian fashion.
92
         */
93
       private final int[] solution;
94
       private final int numberOfMoves;
       private final int numberOfPushes;
96
97
       public SolverReport(SolverParameters parameters,
98
                             ISolverStatistics statistics,
                             List<State> states,
100
                             String status) {
101
            this.parameters = Objects.requireNonNull(parameters);
102
            this.statistics = Objects.requireNonNull(statistics);
            this.status = Objects.requireNonNull(status);
104
105
            if (status.equals(SOLUTION_FOUND)) {
                if (states == null) {
107
                    throw new IllegalArgumentException("SolverStatus is SOLUTION_FOUND." +
108
                             "You must give the solution");
109
                }
111
                SolutionBuilder builder = createFullSolution(states);
112
113
                numberOfPushes = builder.getNumberOfPushes();
114
                numberOfMoves = builder.getNumberOfMoves();
                solution = builder.getSolution();
116
            } else {
117
                numberOfMoves = -1;
                numberOfPushes = -1;
119
                solution = null;
120
            }
121
       }
123
       private SolverReport(SolverParameters parameters,
124
                              ISolverStatistics statistics,
125
                              String status,
                              SolutionBuilder builder) {
127
            this.parameters = Objects.requireNonNull(parameters);
128
            this.statistics = Objects.requireNonNull(statistics);
            this.status = Objects.requireNonNull(status);
130
131
            if (status.equals(SOLUTION_FOUND)) {
132
                numberOfPushes = builder.getNumberOfPushes();
133
                numberOfMoves = builder.getNumberOfMoves();
                solution = builder.getSolution();
135
            } else {
136
                numberOfMoves = -1;
                numberOfPushes = -1;
138
                solution = null;
139
            }
140
       }
```

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

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

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

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

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

```
return parameters.getLevel();
444
        }
445
446
447
        /**
448
         * Returns the pack of the level that was given to the solver
449
450
         * Creturn the pack of the level that was given to the solver
         */
452
        public Pack getPack() {
453
            return parameters.getLevel().getPack();
454
        }
456
457
         * Contains all differences between two states except the old player position.
458
         * Oparam playerDest player destination
460
         * Oparam crate old crate position
461
         * Oparam crateDest crate destination
462
         */
        private record StateDiff(TileInfo playerDest, TileInfo crate, TileInfo crateDest) {}
464
465
466
         * An object to iterate over a solution in forward and backward order.
467
468
        public class SolutionIterator implements ListIterator<Move> {
469
            /**
471
             * Position in the array
472
             */
473
            private int arrayPos;
475
476
             * Position in solution[arrayPos]
477
             */
478
            private int bitPos;
479
480
            private int move;
481
            private int push;
483
484
             * Oreturn read the next bit
485
            private int readNext() {
487
                 int bit = (solution[arrayPos] >> bitPos) & Ob1;
488
489
                 bitPos++;
490
                 if (bitPos == 32) {
491
                     bitPos = 0;
492
                     arrayPos++;
493
                 }
494
495
                 return bit;
496
            }
497
498
            /**
499
             * Oreturn read the previous bit
500
            private int readPrevious() {
502
                 bitPos--;
503
                 if (bitPos < 0) {</pre>
504
                     bitPos = 31;
505
```

```
arrayPos--;
506
                 }
507
508
                 return (solution[arrayPos] >> bitPos) & Ob1;
             }
510
511
512
             @Override
             public boolean hasNext() {
514
                 return move < numberOfMoves;</pre>
515
516
             @Override
518
             public Move next() {
519
                 if (!hasNext()) {
520
                      throw new NoSuchElementException();
522
523
                 int first = readNext();
524
                 int second = readNext();
                 int third = readNext();
526
527
                 int value = (third << 2) | (second << 1) | first;</pre>
                 Move move = Move.values()[value];
530
531
                 this.move++;
532
                 if (move.moveCrate()) {
533
                      push++;
534
535
                 return move;
537
             }
538
539
             @Override
540
             public boolean hasPrevious() {
541
                 return move > 0;
542
543
             @Override
545
             public Move previous() {
546
                 if (!hasPrevious()) {
547
                      throw new NoSuchElementException();
                 }
549
550
                 int third = readPrevious();
551
                 int second = readPrevious();
                 int first = readPrevious();
553
554
                 int value = (third << 2) | (second << 1) | first;</pre>
556
                 Move move = Move.values()[value];
557
558
                 this.move--;
559
                 if (move.moveCrate()) {
560
                      push--;
561
562
                 return move;
564
             }
565
566
             @Override
```

```
public int nextIndex() {
568
                 return move;
569
570
571
             @Override
572
             public int previousIndex() {
573
                 return move - 1;
574
576
             public void reset() {
577
                 move = 0;
578
                 arrayPos = 0;
                 bitPos = 0;
580
581
             @Override
             public void remove() {
584
                 throw new UnsupportedOperationException();
585
586
             @Override
588
            public void set(Move move) {
589
                 throw new UnsupportedOperationException();
592
             @Override
593
             public void add(Move move) {
                 throw new UnsupportedOperationException();
595
596
597
             public int getMoveCount() {
                 return move;
599
600
601
             public int getPushCount() {
602
                 return push;
603
604
        }
605
607
         * A convenience object to convert a list of move to a solution array.
608
609
        private static class SolutionBuilder {
610
611
            private int[] solution;
612
613
             private int arrayPos;
614
             private int bitPos;
615
616
             private int numberOfMoves;
617
             private int numberOfPushes;
618
619
             public SolutionBuilder(int estimatedNumberOfMove) {
620
                 solution = new int[computeArraySize(estimatedNumberOfMove)];
621
623
             private void write(int bit) {
624
                 solution[arrayPos] = (bit & Ob1) << bitPos | solution[arrayPos];</pre>
626
                 bitPos++;
627
                 if (bitPos == 32) {
628
                     bitPos = 0;
```

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

#### FESS0Solver

```
package fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
import fr.valax.sokoshell.solver.collections.SolverCollection;
import fr.valax.sokoshell.solver.heuristic.GreedyHeuristic;
import fr.valax.sokoshell.solver.heuristic.Heuristic;
import java.util.PriorityQueue;
```

```
public class FESSOSolver extends AbstractSolver<FESSOSolver.FESSOState> {
12
       private Heuristic heuristic;
13
       private int lowerBound;
15
       public FESSOSolver() {
16
           super("fess0");
17
19
       @Override
20
       protected void init(SolverParameters parameters) {
21
           heuristic = new GreedyHeuristic(board);
           toProcess = new SolverPriorityQueue();
23
24
       Onverride
       protected void addInitialState(Level level) {
27
           CorralDetector detector = board.getCorralDetector();
28
           State s = level.getInitialState();
29
           board.addStateCrates(s);
31
           detector.findCorral(board, s.playerPos() % level.getWidth(), s.playerPos() /
32

→ level.getWidth());
           board.removeStateCrates(s);
33
34
           lowerBound = heuristic.compute(s);
35
           FESSOState state = new FESSOState(s, 0, lowerBound,

→ detector.getRealNumberOfCorral(), countPackedCrate(s));

38
           toProcess.addState(state);
       }
40
41
       @Override
42
       protected void addState(TileInfo crate, TileInfo crateDest, Direction pushDir) {
           if (checkDeadlockBeforeAdding(crate, crateDest, pushDir)) {
45
           }
46
           final int i = board.topLeftReachablePosition(crate, crateDest);
48
           // The new player position is the crate position
49
           FESSOState s = toProcess.cachedState().child(i, crate.getCrateIndex(),
50

    crateDest.getIndex());
           s.setHeuristic(heuristic.compute(s));
51
           s.setConnectivity(board.getCorralDetector().getRealNumberOfCorral());
52
           s.setPacking(countPackedCrate(s));
53
           if (processed.add(s)) {
55
               toProcess.addState(s);
56
           }
57
       }
58
59
       private int countPackedCrate(State state) {
60
           int nPacked = 0;
61
           for (int crate : state.cratesIndices()) {
               TileInfo tile = board.getAt(crate);
63
               if (tile.isTarget() || tile.isCrateOnTarget()) {
64
                   nPacked++;
               }
           }
67
           return nPacked;
```

```
}
70
71
       @Override
72
       public int lowerBound() {
73
           return lowerBound;
74
75
76
       private static class SolverPriorityQueue extends PriorityQueue<FESSOState>
                implements SolverCollection<FESSOState> {
78
79
            private FESSOState cachedState;
80
            @Override
82
            public void addState(FESSOState state) {
83
                offer(state);
            }
86
            @Override
87
            public FESSOState popState() {
88
                return poll();
90
91
            @Override
            public FESSOState peekState() {
93
                return peek();
94
95
            @Override
97
            public FESSOState peekAndCacheState() {
98
                cachedState = popState();
99
                return cachedState;
            }
101
102
            @Override
103
            public FESSOState cachedState() {
104
                return cachedState;
105
106
       }
107
       protected static class FESSOState extends WeightedState implements
109
        110
            private int connectivity;
            private int packing;
112
113
            public FESSOState(int playerPos, int[] cratesIndices, int hash, State parent, int
114
                cost, int heuristic) {
                super(playerPos, cratesIndices, hash, parent, cost, heuristic);
115
116
117
            public FESSOState(State state, int cost, int heuristic, int connectivity, int
118
               packing) {
                super(state, cost, heuristic);
119
                this.connectivity = connectivity;
120
                this.packing = packing;
            }
122
123
            @Override
            public FESSOState child(int newPlayerPos, int crateToMove, int crateDestination) {
125
                return new FESSOState(super.child(newPlayerPos, crateToMove,
126
                cost(), 0, 0, 0);
```

```
}
129
            public int getConnectivity() {
130
                 return connectivity;
131
            }
132
133
            public void setConnectivity(int connectivity) {
134
                 this.connectivity = connectivity;
136
137
            public int getPacking() {
138
                 return packing;
            }
140
141
            public void setPacking(int packing) {
142
                 this.packing = packing;
144
145
            @Override
146
            public int compareTo(FESSOState o) {
                 // compare in reverse order because
148
                 // java PriorityQueue is a min-queue
149
                 return compare(o, this);
            }
152
            private static int compare(FESSOState a, FESSOState b) {
153
                 // -1 if this < o
                 // 0 if this = o
155
                 // 1 if this > o
156
                 if (a.packing > b.packing) {
157
                     return 1; // we want to maximize packing
                 } else if (a.packing < b.packing) {</pre>
159
                     return -1;
160
                 } else {
161
                     if (a.connectivity < b.connectivity) {</pre>
162
                          return 1; // we want to minimize connectivity
163
                     } else if (a.connectivity > b.connectivity) {
164
                         return -1;
165
                     } else {
                          return Integer.compare(a.weight(), b.weight());
167
168
                 }
169
            }
        }
171
172
```

### **SolverParameters**

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

import fr.poulpogaz.json.IJsonReader;
import fr.poulpogaz.json.IJsonWriter;
import fr.poulpogaz.json.JsonException;
import fr.valax.sokoshell.SokoShell;

import java.io.IOException;
import java.util.*;

/**

* A collection of {@link SolverParameter} plus the name of the solver used and the level

to

* solve. {@link Solver} known which level to solve thanks to this object
```

```
*/
   public class SolverParameters {
15
16
       private final String solverName;
17
       private final Level level;
18
       private final Map<String, SolverParameter> parameters;
19
20
       public SolverParameters(String solverName, Level level) {
           this(solverName, level, null);
22
23
24
       public SolverParameters(String solverName, Level level, List<SolverParameter>
       → parameters) {
           this.solverName = Objects.requireNonNull(solverName);
26
           this.level = Objects.requireNonNull(level);
27
           if (parameters == null) {
29
                this.parameters = Map.of();
30
           } else {
31
                this.parameters = new HashMap<>();
33
                for (SolverParameter p : parameters) {
34
                    this.parameters.put(p.getName(), p);
36
           }
37
       }
38
40
        * Oparam param parameter name
41
        * Oreturn the parameter named param
42
        */
       public SolverParameter get(String param) {
44
           return parameters.get(param);
45
       }
46
47
       /**
49
        * Oparam param name of the parameter
50
        * Oreturn argument of parameter param or default value
        * Oparam <T> type of the argument
52
        * Othrows ClassCastException if the argument can't be cast to a T
53
        */
54
       @SuppressWarnings("unchecked")
       public <T> T getArgument(String param) {
56
           SolverParameter p = parameters.get(param);
57
           if (p == null) {
                throw new NoSuchElementException("No such parameter: " + param);
60
61
           return (T) p.getOrDefault();
63
       }
64
65
       /**
        * Oreturn all parameters
67
68
       public Collection<SolverParameter> getParameters() {
69
           return parameters.values();
       }
71
72
73
       * Oreturn the level to solve
```

```
*/
75
       public Level getLevel() {
76
            return level;
77
        }
78
79
        /**
80
         * Oreturn the name of the solver used
81
        */
       public String getSolverName() {
83
            return solverName;
84
85
87
       public void append(IJsonWriter jw) throws JsonException, IOException {
88
            jw.beginObject();
            jw.field("solver", solverName);
91
            for (Map.Entry<String, SolverParameter> param : parameters.entrySet()) {
92
                if (param.getValue().hasArgument()) {
93
                    jw.key(param.getKey());
                    param.getValue().toJson(jw);
95
                }
96
            }
98
            jw.endObject();
99
        }
100
101
        public static SolverParameters fromJson(IJsonReader jr, Level level) throws
102
            JsonException, IOException {
            jr.beginObject();
103
            String solverName = jr.assertKeyEquals("solver").nextString();
105
            Solver solver = SokoShell.INSTANCE.getSolver(solverName);
106
            if (solver == null) {
107
                throw new IOException("No such solver: " + solverName);
108
            }
109
110
            List<SolverParameter> parameters = solver.getParameters();
111
            while (!jr.isObjectEnd()) {
                String key = jr.nextKey();
113
114
                SolverParameter parameter = parameters.stream()
                         .filter((s) -> s.getName().equals(key))
                         .findFirst()
117
                         .orElseThrow(() -> new IOException("No such parameter: " + key));
118
119
                parameter.fromJson(jr);
120
            }
121
122
            jr.endObject();
124
            return new SolverParameters(solverName, level, parameters);
125
       }
126
   }
```

## FreezeDeadlockDetector

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

import fr.valax.sokoshell.solver.board.Board;
import fr.valax.sokoshell.solver.board.Direction;
import fr.valax.sokoshell.solver.board.tiles.Tile;
```

```
import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   public class FreezeDeadlockDetector {
       // http://www.sokobano.de/wiki/index.php?title=How_to_detect_deadlocks
10
       public static boolean checkFreezeDeadlock(Board board, State state) {
11
           int[] crates = state.cratesIndices();
12
           for (int crate : crates) {
14
               TileInfo info = board.getAt(crate);
15
16
               if (checkFreezeDeadlock(info)) {
                    return true;
18
               }
19
           }
20
           return false;
22
23
24
       public static boolean checkFreezeDeadlock(TileInfo crate) {
           return crate.isCrate() &&
26
                    checkFreezeDeadlockRec(crate, Direction.LEFT) &&
27
                    checkFreezeDeadlockRec(crate, Direction.UP);
       }
30
       private static boolean checkFreezeDeadlockRec(TileInfo crate) {
31
           return checkFreezeDeadlockRec(crate, Direction.LEFT) &&
32
                    checkFreezeDeadlockRec(crate, Direction.UP);
33
       }
34
35
       private static boolean checkFreezeDeadlockRec(TileInfo current, Direction axis) {
           boolean deadlock = false;
37
38
           TileInfo left = current.adjacent(axis);
39
           TileInfo right = current.adjacent(axis.negate());
           if (left.isWall() || right.isWall()) { // rule 1
42
               deadlock = true;
43
           } else if (left.isDeadTile() && right.isDeadTile()) { // rule 2
45
               deadlock = true;
46
47
           } else { // rule 3
               Tile oldCurr = current.getTile();
49
               current.setTile(Tile.WALL);
50
               if (left.anyCrate()) {
52
                    deadlock = checkFreezeDeadlockRec(left);
53
54
               if (!deadlock && right.anyCrate()) {
56
                    deadlock = checkFreezeDeadlockRec(right);
57
                current.setTile(oldCurr);
60
           }
61
62
           return deadlock;
       }
64
   }
65
```

#### **ISolverStatistics**

```
package fr.valax.sokoshell.solver;
   import fr.valax.sokoshell.utils.PrettyTable;
   import fr.valax.sokoshell.utils.Utils;
   import java.io.PrintStream;
6
   import java.io.Serializable;
   /**
   * An object that contains various statistics about a solution, including
10
   * time start and end, number of node explored and queue size at a specific instant
11
   public interface ISolverStatistics extends Serializable {
13
14
        * Returns the time in millis when the solver was started
17
       * @return the time in millis when the solver was started
18
       */
       long timeStarted();
20
21
22
       * Returns the time in millis when the solver stopped running
        * @return the time in millis when the solver stopped running
25
26
       long timeEnded();
27
28
29
       * Returns the time used by the solver to solve a level
       * Oreturn the run time in millis
32
       */
33
       default long runTime() {
34
           return timeEnded() - timeStarted();
       }
36
37
       * Returns the total number of state explored by the solver.
       * If the solver doesn't use State or the {@link Tracker}
40
        * doesn't compute this property, implementations can return
41
        * a negative number
        * Oreturn total number of state explored
44
45
       int totalStateExplored();
48
       * Oreturn number of state explored per seconds or -1
49
       long stateExploredPerSeconds();
51
52
53
       * @return average queue size or -1
55
       int averageQueueSize();
56
57
       /**
       * @return lower bound or -1
59
```

```
int lowerBound();
62
63
        * Print statistics to out.
         * Oparam out standard output stream
66
         * Oparam err error output stream
67
         * Creturn an optional table containing statistics
69
       default PrettyTable printStatistics(PrintStream out, PrintStream err) {
70
            out.printf("Started at %s. Finished at %s. Run time: %s%n",
71
                    Utils.formatDate(timeStarted()),
                    Utils.formatDate(timeEnded()),
73
                    Utils.prettyDate(runTime()));
74
            return null;
       }
77
78
79
        * Basic implementation of {@link ISolverStatistics} then just
        * save time started and time ended
81
         */
82
       record Basic(long timeStarted, long timeEnded) implements ISolverStatistics {
84
            @Override
85
            public int totalStateExplored() {
86
                return -1;
87
            }
89
            @Override
90
            public long stateExploredPerSeconds() {
                return -1;
92
93
94
            @Override
            public int averageQueueSize() {
                return -1;
97
98
            @Override
100
            public int lowerBound() {
101
                return -1;
            }
       }
104
   }
105
```

# Pack

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

import fr.poulpogaz.json.JsonException;
import fr.poulpogaz.json.JsonPrettyWriter;
import fr.poulpogaz.json.JsonReader;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Collections;
import java.util.List;
import java.util.List;
import java.util.Objects;
import java.util.zip.GZIPInputStream;
```

```
import java.util.zip.GZIPOutputStream;
16
17
    * A pack is a collection of levels with a name and an author
19
   public final class Pack {
20
21
        * Some pack doesn't have a name while it is required by {@link
23

    fr.valax.sokoshell.SokoShell}.

        * So pack without a name as named as following: 'Unnamed[I]' where I is an integer
24
   \hookrightarrow which is increased
        * each time an unnamed pack is created. This 'I' is the variable below
25
        */
26
       private static int unnamedIndex = 0;
27
29
       private final String name;
30
       private final String author;
31
       private final List<Level> levels;
33
       private Path sourcePath;
34
       public Pack(String name, String author, List<Level> levels) {
36
           if (name == null) {
37
                this.name = "Unnamed[" + unnamedIndex + "]";
38
                unnamedIndex++;
           } else {
40
                this.name = name;
41
42
           this.author = author;
44
           this.levels = Collections.unmodifiableList(levels);
45
46
           for (Level level : this.levels) {
47
                level.pack = this;
           }
49
       }
50
       public void writeSolutions(Path out) throws IOException, JsonException {
52
           if (out == null) {
53
                out = Path.of(sourcePath.toString() + ".solutions.json.gz");
54
56
           boolean write = false;
57
           for (Level level : levels) {
                if (level.hasReport()) {
                    write = true;
60
61
           }
63
           if (!write) {
64
               return;
65
           }
           try (OutputStream os = Files.newOutputStream(out, StandardOpenOption.CREATE,
68
               StandardOpenOption.TRUNCATE_EXISTING)) {
               BufferedWriter bw = new BufferedWriter(
                        new OutputStreamWriter(new GZIPOutputStream(os)));
70
71
                JsonPrettyWriter jpw = new JsonPrettyWriter(bw);
72
```

```
jpw.beginObject();
74
                 if (name != null) {
75
                     jpw.field("pack", name);
76
                 } else {
77
                     jpw.nullField("pack");
78
                 }
79
                 if (author != null) {
80
                     jpw.field("author", author);
                 } else {
82
                     jpw.nullField("author");
83
84
                 for (Level level : levels) {
86
                     if (level.hasReport()) {
87
                          jpw.key(String.valueOf(level.getIndex()));
                          jpw.beginArray();
90
91
                          level.writeSolutions(jpw);
92
                          jpw.endArray();
94
                     }
95
                 }
97
                 jpw.endObject();
98
                 jpw.close();
99
            }
100
        }
101
102
        public void readSolutions(Path in) throws IOException, JsonException {
103
            if (Files.notExists(in)) {
                 return;
105
106
107
            try (InputStream is = Files.newInputStream(in)) {
108
                 BufferedReader br = new BufferedReader(
109
                         new InputStreamReader(new GZIPInputStream(is)));
110
111
                 JsonReader jr = new JsonReader(br);
113
                 jr.beginObject();
114
                 jr.assertKeyEquals("pack");
                 String pack;
117
                 if (jr.hasNextString()) {
118
                     pack = jr.nextString();
119
                 } else {
120
                     jr.nextNull();
121
                     pack = null;
122
                 }
124
                 jr.assertKeyEquals("author");
125
                 String author;
126
                 if (jr.hasNextString()) {
127
                     author = jr.nextString();
128
                 } else {
129
                     jr.nextNull();
130
                     author = null;
                 }
132
133
                 if (Objects.equals(pack, name) && Objects.equals(author, this.author)) {
134
135
```

```
while (!jr.isObjectEnd()) {
136
                          int level = Integer.parseInt(jr.nextKey());
137
138
                          Level 1 = levels.get(level);
                          jr.beginArray();
140
141
                          while (!jr.isArrayEnd()) {
142
                              jr.beginObject();
                              1.addSolverReport(SolverReport.fromJson(jr, 1));
144
                              jr.endObject();
145
146
                          jr.endArray();
148
                     }
149
150
                     jr.endObject();
152
                 jr.close();
153
            }
154
        }
156
157
         * Returns the name of the pack
         * Oreturn the name of the pack
160
161
        public String name() {
162
            return name;
163
164
165
         * Returns the author of the pack
167
168
         * @return pack's author
169
         */
170
        public String author() {
171
            return author;
172
173
175
         * Returns all levels that are in this pack
176
177
         * @return levels of this pack
179
        public List<Level> levels() {
180
            return levels;
181
        }
182
183
184
         * Returns the level at the specified index
186
         * Oparam index the index of the level
187
         * Oreturn the level at the specified index
188
         * Othrows IndexOutOfBoundsException if the index is out of range
189
190
        public Level getLevel(int index) {
191
            return levels.get(index);
192
        }
194
195
         * Returns the number of level in this pack
196
197
```

```
* @return the number of level in this pack
198
199
        public int nLevels() {
200
            return levels.size();
201
202
203
204
         * Returns the location of the file describing this pack. This is used for writing
       solutions
206
         * Oreturn the location of the file describing this pack
207
         * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
209
        public Path getSourcePath() {
210
            return sourcePath;
211
213
214
         * Sets the location of the file describing this pack. This is used for writing
215
       solutions
216
         * @see fr.valax.sokoshell.readers.Reader#read(Path, boolean)
217
        public void setSourcePath(Path sourcePath) {
            this.sourcePath = sourcePath;
220
221
```

#### Hotspots

```
package fr.valax.sokoshell.solver;
   import fr.valax.sokoshell.solver.board.Board;
   import fr.valax.sokoshell.solver.board.Direction;
   import fr.valax.sokoshell.solver.board.tiles.TileInfo;
   import java.util.ArrayDeque;
   import java.util.Arrays;
   import java.util.HashSet;
   import java.util.Queue;
10
11
   public class Hotspots {
12
13
       private final Board board;
14
15
       /**
        * if hotspot[X][Y] is true then if there is a crate at Y and at X,
17
        * Y blocks X to be pushed to at least one target
18
        */
19
       private final boolean[][] hotspot;
20
21
       // Variables used by countAccessibleTargets
22
       private ReachableTiles reachable;
23
       // accessible[x] is true if at x there is target and the target is push-accessible
       private boolean[] accessible;
25
       private Queue<State> toVisit;
26
       private HashSet<State> visited;
27
28
       public Hotspots(Board board) {
29
           this.board = board;
30
           int s = board.getWidth() * board.getHeight();
           this.hotspot = new boolean[s][s];
32
```

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

```
State s = new State(player, baseCrate);
95
                     toVisit.add(s);
96
                     visited.add(s);
97
                 }
            }
99
100
            if (baseCrate.isCrateOnTarget()) {
101
                 accessible[baseCratePos] = true;
103
104
            baseCrate.removeCrate();
105
            while (!toVisit.isEmpty()) {
107
                State s = toVisit.poll();
108
109
                 s.crate().addCrate();
                 reachable.findReachableCases(s.player());
111
112
                for (Direction dir : Direction.VALUES) {
113
                     TileInfo player = s.crate().adjacent(dir.negate());
                     TileInfo crateDest = s.crate().adjacent(dir);
115
116
                     if (crateDest.isSolid() || !reachable.isReachable(player)) {
117
                         continue;
                     }
119
120
                     if (crateDest.isTarget()) {
                         accessible[crateDest.getIndex()] = true;
122
                     }
123
124
                     int playerDest = board.topLeftReachablePosition(s.crate(), crateDest);
126
                     State newState = new State(board.getAt(playerDest), crateDest);
127
                     if (visited.add(newState)) {
128
                         toVisit.add(newState);
129
                     }
130
                 }
131
132
                 s.crate().removeCrate();
            }
134
            baseCrate.addCrate();
135
136
            return countTrue(accessible);
        }
138
139
        private int countTrue(boolean[] array) {
140
            int n = 0;
141
142
            for (int i = 0; i < array.length; i++) {</pre>
143
                 if (array[i]) {
                     n++;
145
                 }
146
            }
147
            return n;
        }
150
151
        public boolean isHotspot(int crate, int blockingCrate) {
152
            return hotspot[crate][blockingCrate];
153
154
155
        private record State(TileInfo player, TileInfo crate) {}
```

157 }