# Writing an unfinished “Baba Is You” solver

Julien Kirch

Version 0.1, 2019-06-02

# Table of Contents

# Part 1: reading levels

Baba Is You is a great puzzle game with a gameplay that's more complex than usual since manipulating the game's rules is part of the game.

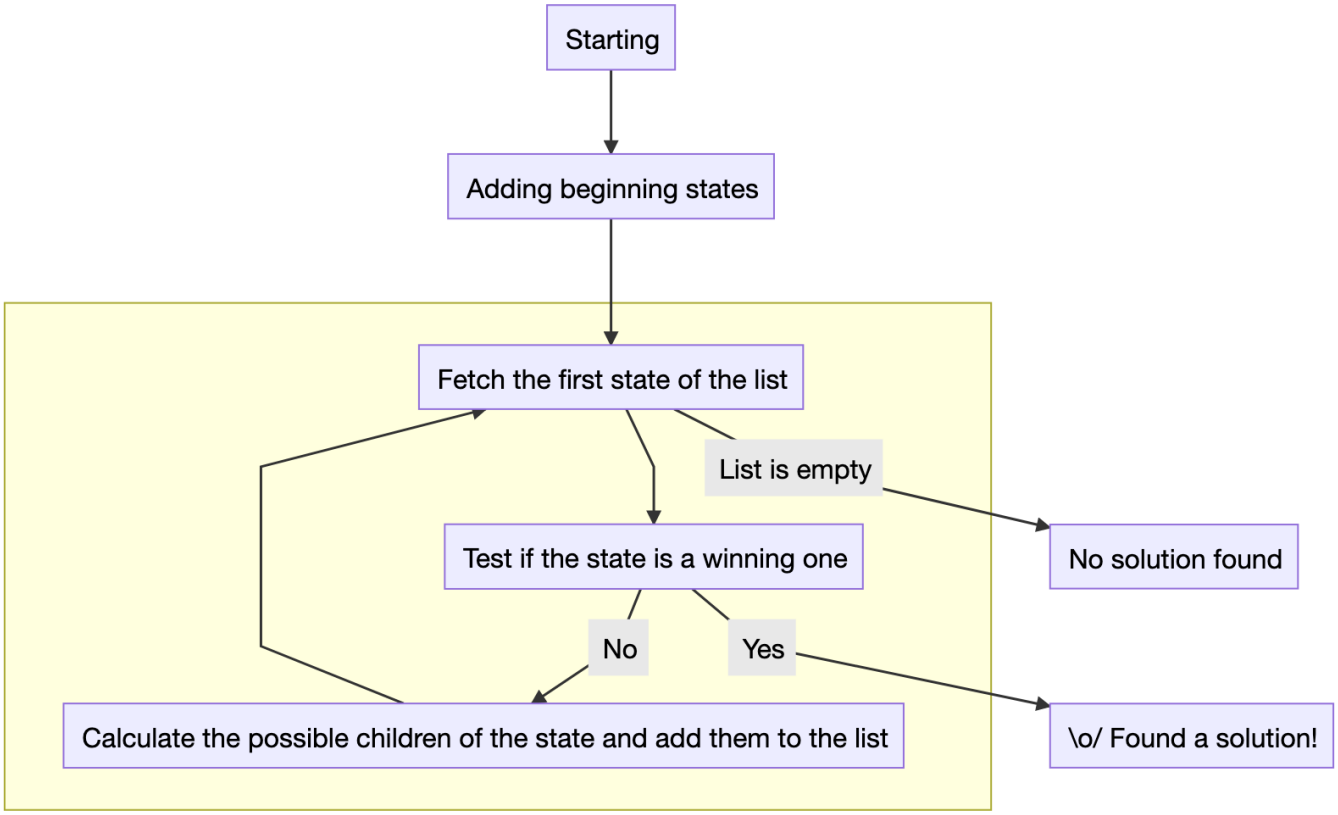
I think it's a good candidate to write a solver for, and to document how I'm doing it.

I'm writing while coding the solver and not as a post-mortem, so expect some bad design and implementation decisions along the way.

A general knowledge of the game (if you don't want to play it you can try watching a "let's play" video like this one) and of Java is required.

A solver often has complex data types, and you end up refactoring the code a lot when trying new ideas. When refactoring, I've found it very helpful to have a language with static types. My tool of choice for this kind of situation is Java. As a nice bonus, the language is fast so less time is spent waiting for the calculations to end.

As I've explained in a previous article (in French), a puzzle solver solver iterates through a loop until it finds a solution.




The first step would be to be able to initialize the loop, which means reading the levels from disk. But before I can get there, I need to setup a project for the code.

## Create the project

First thing is to bootstrap the project.

Now, to create the project:

```
mvn archetype:generate \
-DgroupId=net.archiloque.babaisyousolver \
-DartifactId=babaisyousolver \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.4 \
-DinteractiveMode=false
```

Then add the required files (`LICENSE.txt` …) and push it to GitHub.

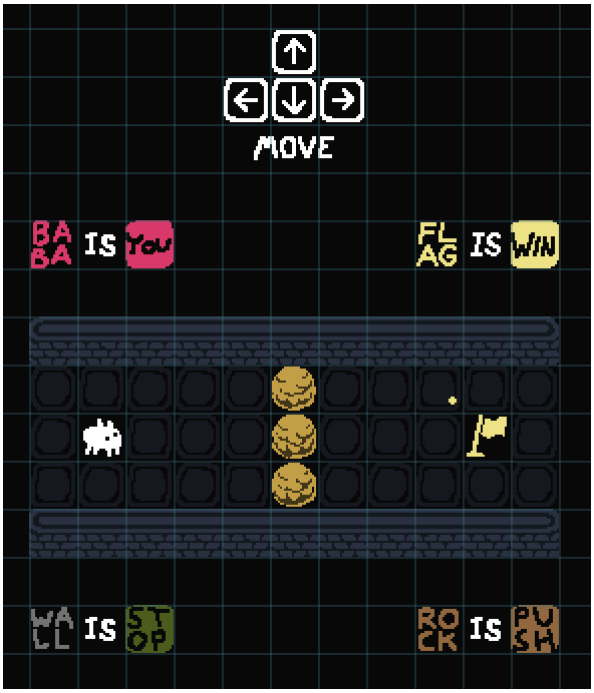
# Read levels

In my previous solvers, I represent levels using a fixed list of elements, each represented by a fixed character.

Judging from the levels I've played in Baba so far, there seems to be an uncommonly large number of possible tiles. For that reason, I'm considering an alternate strategy:

1. Define a list of all the possible tiles, but don't tie them to a specific character
2. For each level, declare a mapping that assigns a character to each of the tiles used in this specific level

The levels are laid out in a square grid. In the starting position, each cell contains at most one item.




*The first level*

Let's represent this level as the following two files:

*levels/00/tiles.txt*

```
b Baba
B Baba text
  empty
f flag
F flag text
I is text
P push text
r rock
R rock text
S stop text
w wall
W wall text
! win text
Y you text
```

*levels/00/content.txt*

```
BIY     FI!

wwwwwwwwww
     r
 b   r   f
     r
wwwwwwwwww

WIS     RIP
```

Note: I'd love to represent the objects as emojis rather than plain letters, to make the level files more visually appealing. Unfortunately, the way macOS handles emojis is not monospace friendly. It breaks the vertical alignment between elements in a level file, making them a pain to edit. That's why I'm sticking with latin characters for now.

Let's declare the tiles needed for the first level. Each tile has two representations: one as a `String` to be referenced by the level description, the other one as an `int` to be referenced by the game:

*Tiles.java*

```
/**
 * All the tiles.
 */
interface Tiles {

  String BABA_STRING = "Baba";
  String BABA_TEXT_STRING = "Baba text";
  String EMPTY_STRING = "empty";
  String FLAG_STRING = "flag";
  String FLAG_TEXT_STRING = "flag text";
```

```
  String IS_TEXT_STRING = "is text";
  String PUSH_TEXT_STRING = "push text";
  String ROCK_STRING = "rock";
  String ROCK_TEXT_STRING = "rock text";
  String STOP_TEXT_STRING = "stop text";
  String WALL_STRING = "wall";
  String WALL_TEXT_STRING = "wall text";
  String WIN_TEXT_STRING = "win text";
  String YOU_TEXT_STRING = "you text";

  String[] ALL_STRINGS = new String[]{
      BABA_STRING,
      BABA_TEXT_STRING,
      EMPTY_STRING,
      FLAG_STRING,
      FLAG_TEXT_STRING,
      IS_TEXT_STRING,
      PUSH_TEXT_STRING,
      ROCK_STRING,
      ROCK_TEXT_STRING,
      STOP_TEXT_STRING,
      WALL_STRING,
      WALL_TEXT_STRING,
      WIN_TEXT_STRING,
      YOU_TEXT_STRING,
  };

  int BABA = 0;
  int BABA_TEXT = BABA + 1;
  int EMPTY = BABA_TEXT + 1;
  int FLAG = EMPTY + 1;
  int FLAG_TEXT = FLAG + 1;
  int IS_TEXT = FLAG_TEXT + 1;
  int PUSH_TEXT = IS_TEXT + 1;
  int ROCK = PUSH_TEXT + 1;
  int ROCK_TEXT = ROCK + 1;
  int STOP_TEXT = ROCK_TEXT + 1;
  int WALL = STOP_TEXT + 1;
  int WALL_TEXT = WALL + 1;
  int WIN_TEXT = WALL_TEXT + 1;
  int YOU_TEXT = WIN_TEXT + 1;
}
```

I could see this file becoming a pain to maintain later on. If it does, I will put its content into a configuration file and generate the Java code from that. For now I prefer to keep it simple.

You might be surprised at my use of `int`s rather than enum types for this kind of data. I have experimented with both in previous solvers. I now prefer `int`s for the following reasons:

- They use less memory. When you have to store many levels, this makes a big difference.
- You can use the `int` as an index into an array to retrieve data. This is much faster than using a `Hash`.

Using `int`s also has its downsides. You lose some type checking, and the code becomes a little harder to read. For me, it's an acceptable tradeoff.

Let's now draft the interface for reading level files:

*LevelReader1.java*

```
final class LevelReader {

  private LevelReader() {
  }

  private static final String TILES_FILES = "tiles.txt";
  private static final String CONTENT_FILES = "content.txt";

  /**
   * Read a level from a directory.
   */
  static @NotNull LevelReaderResult readLevel(
      @NotNull Path levelDirectory)
      throws IOException {
    return readContent(levelDirectory, readTiles(levelDirectory));
  }

  private final static Pattern TILES_REGEX =
      Pattern.compile("^(.{1}) (.+)$");

  /**
   * Read the tiles declaration
   * from the {@link LevelReader#TILES_FILES} file
   */
  static @NotNull Map<Character, Integer> readTiles(
      @NotNull Path levelDirectory) throws IOException {
    // @TODO probably add some code here
    return new HashMap<>();
  }

  /**
   * Read the file content
   * from the {@link LevelReader#CONTENT_FILES} file
   * relying the declared tiles
   */
  static @NotNull LevelReaderResult readContent(
      @NotNull Path levelDirectory,
      @NotNull Map<Character,
```

```
            Integer> tiles) throws IOException {
      // @TODO probably add some code here
      return null;
    }


    static final class LevelReaderResult {

      final int width;
      final int height;
      final @NotNull int[] content;

      LevelReaderResult(
          int width,
          int height,
          @NotNull int[] content) {
        this.width = width;
        this.height = height;
        this.content = content;
      }
    }
}
```

Calling `LevelReader#readLevel` will return a struct-like `LevelReader.LevelReaderResult` containing the data in a ready to use format, I'll keep `Level` for the real object.

I represent two dimensional data as flat arrays rather than arrays of arrays: they are simpler to copy and faster to access.

It's natural to represent a position in such an array as a single integer. Compared to a more traditional representation as a pair of coordinates, this saves having to create an object each time I want to return a position from a function or pass one as an argument, specially in Java were there are no struct and only full-fledged classes.

The code to convert between a line/column representation and a position in a flat array is straightforward:

```
int position = (indexLine * levelWidth) + indexColumn;

int indexLine = position / levelWidth;
int indexColumn = position % levelWidth;
```

One final note of interest: to make up for Java's lacking native handling of nullability, I make extensive use of JetBrains's annotations. I find them to be an acceptable substitute as long as I use Idea.

# Just add code

Let's write a few test cases (function bodies elided for brevity since the code is rather boring):

*LevelReaderTest.java*

```
class LevelReaderTest {

  @Test void readTilesOK(){}

  @Test void readTilesFileNotFound(){}

  @Test void readTilesTilesNotFound(){}

  @Test void readTilesInvalidSyntax(){}

  @Test void readContentOK(){}

  @Test void readContentFileNotFound(){}

  @Test void readContentInvalidLineLength(){}

  @Test void readContentUnknownTile(){}
}
```

There are a lot tests covering error cases. That's because I hate having to wade through my code with a debugger to find out what's wrong with a level file. I'd rather take the time to generate error messages that are as explicit as possible.

Then fill the content of the reader:

*LevelReader.java*

```
/**
  * Read the tiles declaration
  * from the {@link LevelReader#TILES_FILES} file
  * @return a map linking each char to the {@link Tiles} id
  */
static @NotNull Map<Character, Integer> readTiles(
    @NotNull Path levelDirectory
) throws IOException {
  Path elementsFile = getFile(levelDirectory, TILES_FILES);
  List<String> tilesContent = Files.readAllLines(elementsFile);

  Map<Character, Integer> result = new HashMap<>();

  for (String tileLine : tilesContent) {
    Matcher m = TILES_REGEX.matcher(tileLine);
    if (!m.find()) {
      throw new IllegalArgumentException(
          "Bad tile declaration [" + tileLine + "]");

    }
```

```
    Character character = m.group(1).charAt(0);
    String tile = m.group(2);
    int index = findIndexOfTile(tile);
    if (index < 0) {
      throw new IllegalArgumentException(
          "Unknown tile [" + tile + "]");
    }
    result.put(character, index);
  }
  return result;
}

/**
 * Find the index of a {@link Tiles} from its name
 * @param tileName the index or -1 if not found
 */
private static int findIndexOfTile(
    @NotNull String tileName) {
  for (int index = 0; index < Tiles.ALL_STRINGS.length; index++) {
    if (tileName.equals(Tiles.ALL_STRINGS[index])) {
      return index;
    }
  }
  return -1;
}

/**
  * Read the file content
  * from the {@link LevelReader#CONTENT_FILES} file
  * relying the declared tiles
  */
static @NotNull LevelReaderResult readContent(
    @NotNull Path levelDirectory,
    @NotNull Map<Character, Integer> tiles
) throws IOException {
  Path elementsFile = getFile(levelDirectory, CONTENT_FILES);
  List<String> contentContent = Files.readAllLines(elementsFile);

  int height = contentContent.size();
  int width = contentContent.get(0).length();
  int[] content = new int[height * width];

  for (int lineIndex = 0; lineIndex < height; lineIndex++) {
    String contentLine = contentContent.get(lineIndex);
    if (contentLine.length() != width) {
      throw new IllegalArgumentException(
          "[" +
              contentLine +
```

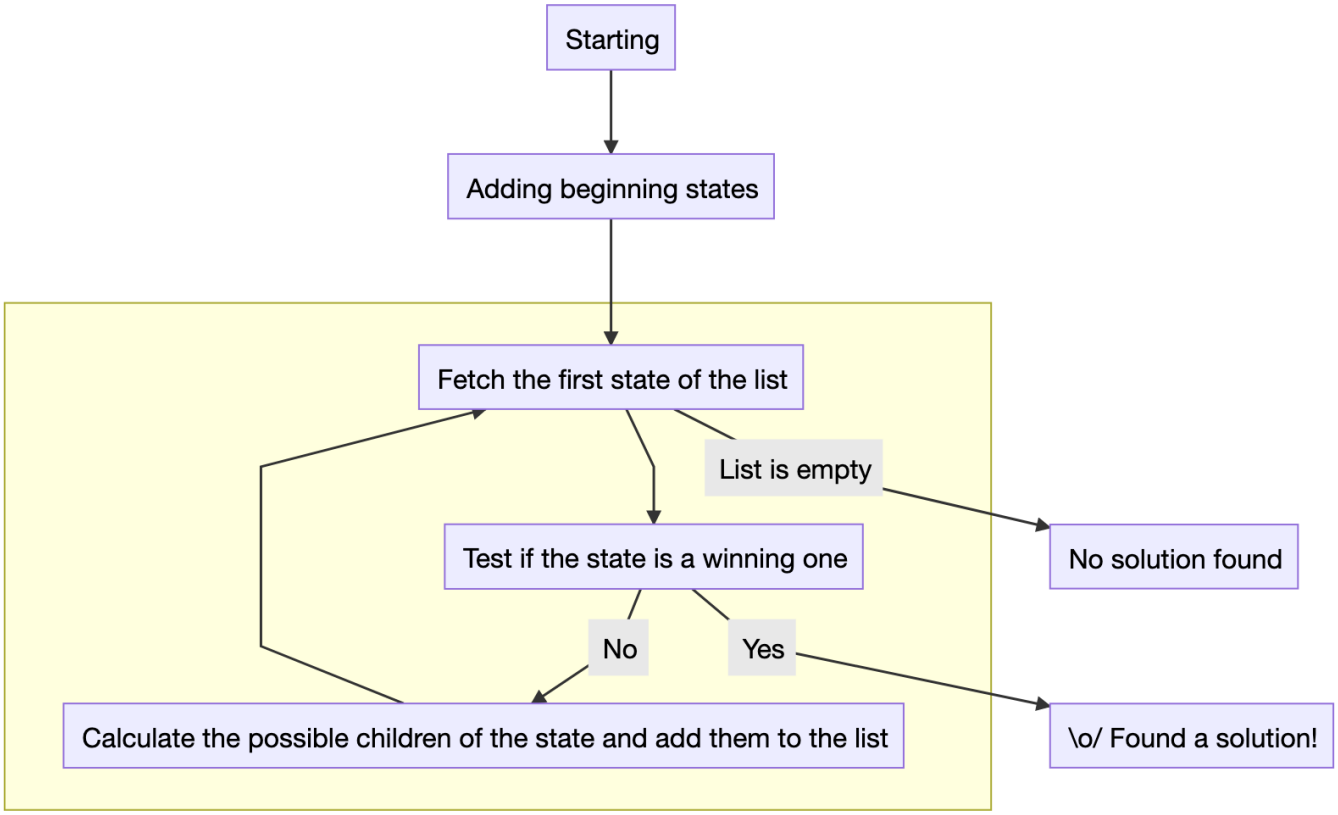
```
                "] is not " +
                width +
                " characters long at line " +
                lineIndex +
                " of " +
                elementsFile.toAbsolutePath());
    }

    for (int columnIndex = 0; columnIndex < width; columnIndex++) {
      char c = contentLine.charAt(columnIndex);

      if (!tiles.containsKey(c)) {
        throw new IllegalArgumentException(
            "Unknown tile [" +
                c +
                "] at line " +
                lineIndex +
                " of " +
                elementsFile.toAbsolutePath());
      }
      int position = (lineIndex * width) + columnIndex;
      content[position] = tiles.get(c);
    }
  }
  return new LevelReaderResult(width, height, content);
}

private static @NotNull Path getFile(
    @NotNull Path directory,
    @NotNull String fileName
) throws FileNotFoundException {
  Path file = directory.resolve(fileName);
  if (!Files.exists(file)) {
    throw new FileNotFoundException(
        file.toAbsolutePath().toString());
  }
  return file;
}
```

Now I have all the code I need to read a level from disk. In the next part I'll start implementing the solving logic.

# Part 2: plumbing

In the previous part I wrote the code to read the levels. While I'd love to jump straight to the solving logic, I need to establish the game loop first. It's plumbing time!

This is fairly generic code, and I was able to port most of it from my previous efforts.




## The level

I'll use a `Level` class to hold all the data related to a game level. For now it will just contain the level data we read from disk. I know from experience that I'll add other data to simplify and speed up the calculations.

*Level.java*

```
class Level {

  final int width;
  final int height;
  final int size;
  final @NotNull int[] content;

  Level(
      int width,
      int height,
      @NotNull int[] content) {
    this.width = width;
    this.height = height;
    this.size = width * height;
    this.content = content;
  }
}
```

My next class, the `State`, represents a state of the game being solved. Just as the `Level`, it starts out as a small class with few attributes (the current status of the game and a reference to the `Level`) and it will become larger as I add features. For instance, this is where I'll handle Baba's dynamically changing rules.

*State.java*

```
class State {

  private final @NotNull Level level;

  /**
   * Probably not the right format, just drafting
   */
  private final @NotNull int[] content;

  State(
      @NotNull Level level,
      @NotNull int[] content) {
    this.level = level;
    this.content = content;
  }

  /**
   * Process the current state
   *
   * @return true if a solution is found
   */
  boolean process() {
    // @TODO probably add some code here
    return false;
  }
}
```

# The list of states

I could represent my list of game states as a FIFO queue to implement a depth-first search or as a stack for breadth-first search.

A FIFO may find a solution faster but the solution may be not be the shortest, a stack will find the solution that requires the lowest number of move, but requires more time and memory since it tends explore more possibilities before finding a solution.

Which works best depends on the game I'm solving. Rather than choose by a fixed criterion, I just start from something that makes sense and tweak it as I go.

I used a FIFO queue for my last project, let's use that again.

Rather than depend on an existing FIFO implementation, I just roll my own. The code is short and it does exactly what I need and nothing more. It's another thing to tweak when a solver hit a speed limit, in my last games this code was the faster.

*FiFoQueue.java*

```
final class FiFoQueue<E> {

  private @Nullable FiFoQueue.QueueElement<E> currentElement;

  FiFoQueue() {
  }

  void add(@NotNull E newElement) {
    currentElement =
        new QueueElement<>(newElement, currentElement);
  }

  @Nullable E pop() {
    if (currentElement != null) {
      E element = currentElement.element;
      currentElement = currentElement.next;
      return element;
    } else {
      return null;
    }
  }

  boolean isEmpty() {
    return currentElement == null;
  }

  private final static class QueueElement<E> {

    private final @NotNull E element;

    private final @Nullable FiFoQueue.QueueElement<E> next;

    QueueElement(
        @NotNull E element,
        @Nullable FiFoQueue.QueueElement<E> next) {
      this.element = element;
      this.next = next;
    }
  }
}
```

I now have all the elements I need to add state list management code to the `Level` class.

*Level.java*

```
private final @NotNull FiFoQueue<State> states =
    new FiFoQueue<>();

void createInitStates() {
  addState(content);
}

void addState(@NotNull int[] content) {
  states.add(new State(this, content));
}

@Nullable State solve() {
  while (true) {
    State state = states.pop();
    if(state == null) {
      return null;
    }
    if (state.process()) {
      return state;
    }
  }
}
```

Calling `State#process` will add some new possible `State`s to the list, so the minimal version of the solver loop is now complete.

# The entry point

Finally, my solver needs an entry point.

*App.java*

```
/**
 * Entry point
 */
public class App {

  private static final SimpleDateFormat DATE_FORMAT =
      new SimpleDateFormat("yyyy-MM-dd kk:mm:ss.SSS");

  public static void main(String[] args)
      throws IOException {
    if (args.length == 0) {
      throw new IllegalArgumentException(
          "A level path should be specified");
    }
```

```
    processLevel(Path.of(args[0]));

  }

  private static void processLevel(
      @NotNull Path path
  ) throws IOException {
    print(path, "Reading level");
    LevelReader.LevelReaderResult levelReaderResult =
        LevelReader.readLevel(path);
    Level level = new Level(
        levelReaderResult.width,
        levelReaderResult.height,
        levelReaderResult.content);
    level.createInitStates();
    print(path, "Solving level");
    long startTime = System.nanoTime();
    State solution = level.solve();
    long stopTime = System.nanoTime();
    String endTime =
        LocalTime.MIN.plusNanos((stopTime - startTime)).
            toString();
    if (solution != null) {
      print(
          path,
          "Solved in " + endTime);
    } else {
      print(
          path,
          "Failed in " + endTime);
    }
  }

  private static void print(
      @NotNull Path path,
      @NotNull String message) {
    System.out.println(
        DATE_FORMAT.format(new Date()) +
            " " +
            path.toAbsolutePath() +
            " " +
            message);
  }
}
```

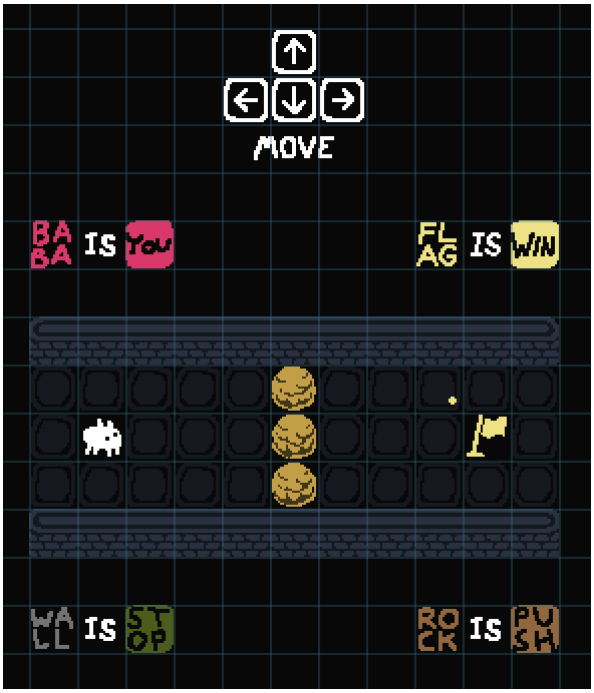
With the workflow completed, I can now run the program run and watch it solve its first level … well, kind of:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Failed in 00:00:00.000008930
```

In the next part I'll put all the plumbing to good use and add some logic to the code.

# Part 3: finding a solution to the first level

Following the second part, I'll deal with the logic to find a solution to the first level.




*Here it is!*

The first level has a feature that will make things much simpler: the rules can't be changed.

This means that fixed rules can be enough to solve this level, and that the "changing rules" feature can be dealt with later.

As my goal is to reach a state where I solved something, I'll first try to find a valid solution with a simplified algo that don't cover all possible cases, then I'll update it to make it compliant with the game rules.

## The game loop

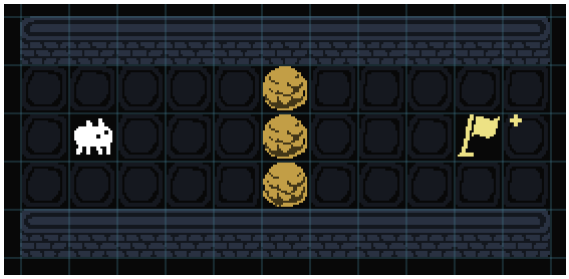
Each turn the solver considers a game state, from it there is *at most* four possible moves: going up, down, right or left.

At most because some of the directions can be blocked by an object or by the border of the level's board.

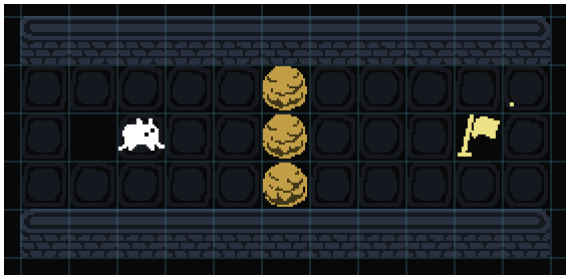
If you can go in a direction, you may reach a winning state, in this case the the solver stops.

If you don't solve the level, you can add this new state to the list, and go on with the main loop.

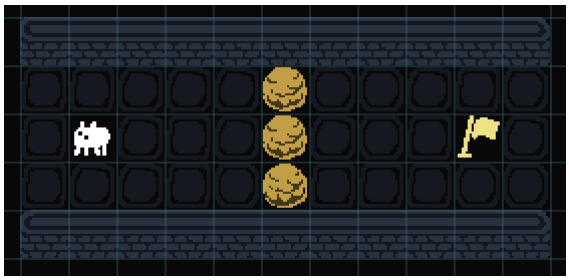
There's a catch: Baba is not a "burning bridge" game where you can't move back. Before checking if a new state is interesting, it must be vetted against the previously met cases to avoid doing the same thing again and again.



*From here*



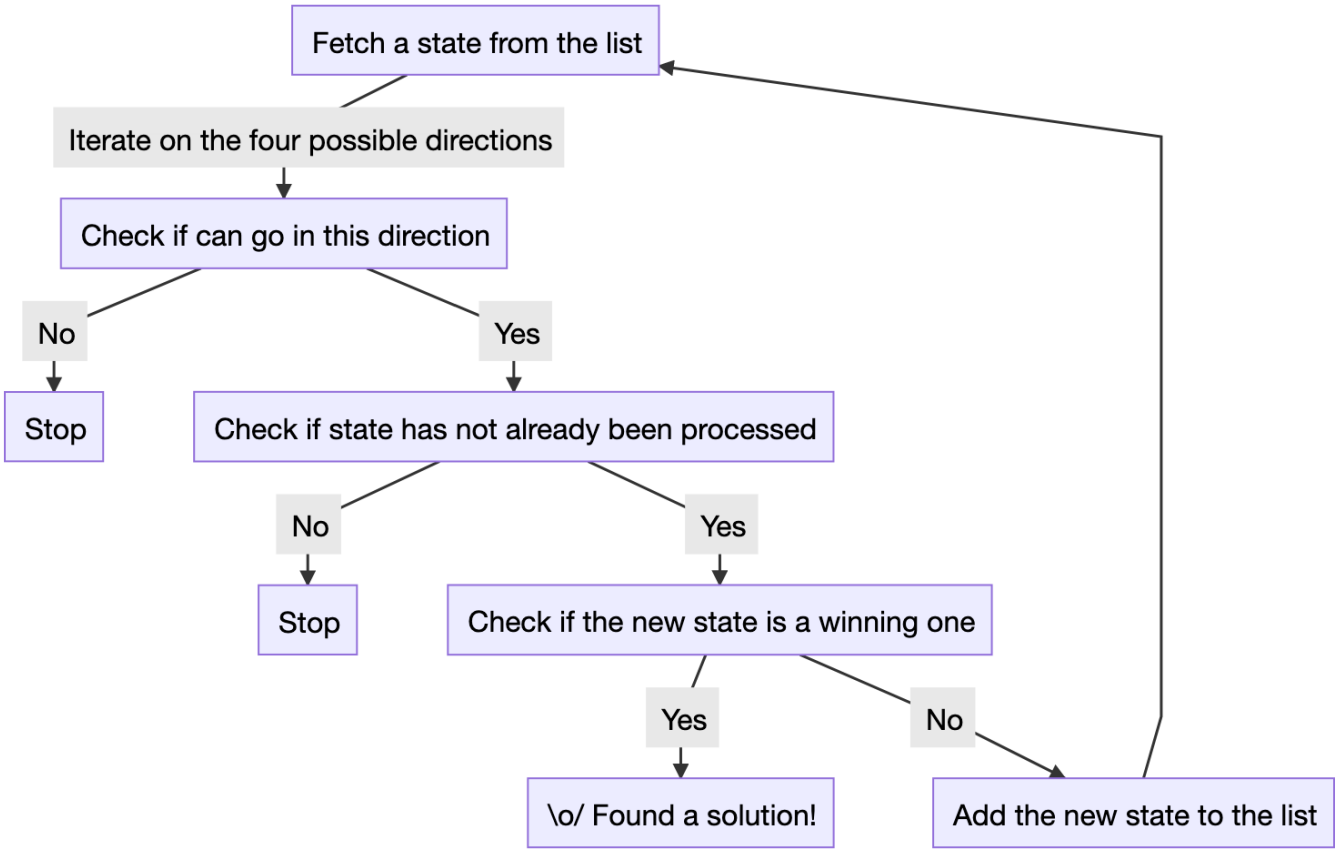
*Baba can go there*



*And then go back*

Without this vetting, Baba could go left, then right, then left, then right… and time would be wasted to deal with states that have already been processed.

So this is the algo to be implemented for each state




# It's alive!

First thing is to detect if a move is possible.

In the further levels of the game, "you" can be several items, but in the first level you only control Baba, so first let's find Baba.

To be able to go in a direction only requires to test if Baba can go in this direction.

First I check if Baba can physically go in this direction from the level point of view.

It requires to be able to specify a direction.

*Direction.java*

```
interface Direction {

  byte UP = 0;
  byte DOWN = 1;
  byte LEFT = 2;
  byte RIGHT = 3;

  char[] VISUAL = new char[]{
      '↑', '↓', '←', '→'
  };
}
```

Then the code:

*State.java*

```
class State {

  private final @NotNull Level level;

  /**
   * Probably not the right format, just drafting
   */
  private final @NotNull int[] content;

  State(
      @NotNull Level level,
      @NotNull int[] content) {
    this.level = level;
    this.content = content;
  }

  /**
   * Process the current state
   *
   * @return true if a solution is found
   */
  boolean process() {
    int babaPosition = findBaba();
    int babaLine = babaPosition / level.width;
    int babaColumn = babaPosition % level.width;

    // Up
    if (babaLine > 0) {
      if (tryToGo(babaPosition, Direction.UP)) {
        return true;
      }
    }

    // Down
    if (babaLine < (level.height - 1)) {
      if (tryToGo(babaPosition, Direction.DOWN)) {
        return true;
      }
    }

    // Left
    if (babaColumn > 0) {
      if (tryToGo(babaPosition, Direction.LEFT)) {
        return true;
      }
    }
```

```
    // Right
    if (babaColumn < (level.width - 1)) {
      if (tryToGo(babaPosition, Direction.RIGHT)) {
        return true;
      }
    }

    return false;
  }

  boolean tryToGo(int currentPosition, byte direction) {
    // @TODO probably add some code here
    return false;
  }

  /**
   * Find the index of the baba position.
   *
   * @return the position or -1 if not found
   */
  int findBaba() {
    for (int i = 0; i < level.size; i++) {
      if (content[i] == Tiles.BABA) {
        return i;
      }
    }
    return -1;
  }
```

And testing it:

*StateAvailableMovementsTest.java*

```
/**
 * Try every positions on a 3*3 level
 * and check where baba can go
 */
class StateAvailableMovementsTest {

  /**
   * {@link State} that records the possible moves
   */
  private static final class StateToTestAvailableMovements extends State {

    private final List<Byte> movements = new ArrayList<>();

    StateToTestAvailableMovements(
```

```
        @NotNull Level level,
        @NotNull int[] content) {
      super(level, content, new byte[0]);
    }

    @Override
    byte[] tryToGo(int currentPosition, byte position) {
      movements.add(position);
      return null;
    }
  }

  private void checkAvailableMovements(
      int babaIndex,
      @NotNull Byte[] movements) {
    int[] levelContent = new int[9];
    Arrays.fill(levelContent, Tiles.EMPTY);
    levelContent[babaIndex] = Tiles.BABA;
    Level level = new Level(3, 3, levelContent);
    StateToTestAvailableMovements state =
        new StateToTestAvailableMovements(level, levelContent);
    state.process();
    assertArrayEquals(movements, state.movements.toArray());
  }

  @Test
  void testAvailableMovement() {
    checkAvailableMovements(0,
        new Byte[]{DOWN, RIGHT});
    checkAvailableMovements(1,
        new Byte[]{DOWN, LEFT, RIGHT});
    checkAvailableMovements(2,
        new Byte[]{DOWN, LEFT});
    checkAvailableMovements(3,
        new Byte[]{UP, DOWN, RIGHT});
    checkAvailableMovements(4,
        new Byte[]{UP, DOWN, LEFT, RIGHT});
    checkAvailableMovements(5,
        new Byte[]{UP, DOWN, LEFT});
    checkAvailableMovements(6,
        new Byte[]{UP, RIGHT});
    checkAvailableMovements(7,
        new Byte[]{UP, LEFT, RIGHT});
    checkAvailableMovements(8,
        new Byte[]{UP, LEFT});
  }

}
```

Then it's time to deal with the content. The behavior depends of the content of the position Baba wants to move to. The code must do three things:

1. test if the movement is valid
2. test if the new position is a winning one
3. apply the change to the the board's content to represent the new state

It would be possible to write each step in separate pass but I find it easier to do it in one time because all three steps depends mostly of the kind of elements on the target cell.

To apply the change I clone the existing state and modify the values that need to be updated.

Pushing a rock is more complex case than the others, so I'll ignore it first:

*State.java*

```
/**
 * Try to go on a direction from a position
 * @return true if a solution has been found
 */
boolean tryToGo(int currentPosition, byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  int[] newContent;
  switch (targetPositionContent) {
    case Tiles.WALL:
      return false;
    case Tiles.EMPTY:
      newContent = content.clone();
      newContent[targetPosition] = Tiles.BABA;
      newContent[currentPosition] = Tiles.EMPTY;
      level.addState(newContent);
      return false;
    case Tiles.ROCK:
      // @TODO implements this
      return false;
    case Tiles.FLAG:
      return true;
    default:
      throw new IllegalArgumentException(
        Integer.toString(targetPositionContent));
  }
}

/**
 * Calculate the index of a position after a move
 */
private int calculatePosition(int position, byte direction) {
```

```
  // Content is stored as a single array one line after another
  switch (direction) {
    case Direction.UP:
      // up: go back one row
      return position - level.width;
    case Direction.DOWN:
      // down: go further one row
      return position + level.width;
    case Direction.LEFT:
      // left : go back one item
      return position - 1;
    case Direction.RIGHT:
      // left : go further one item
      return position + 1;
    default:
      throw new IllegalArgumentException(
        Integer.toString(direction));
  }
}
```

And testing it:

*StateTryToGoTest.java*

```
class StateTryToGoTest {

  private static final class LevelToTestTryToGo extends Level {

    private final List<int[]> states = new ArrayList<>();

    LevelToTestTryToGo(
        int width,
        int height,
        @NotNull int[] content) {
      super(width, height, content);
    }

    @Override
    void addState(@NotNull int[] content) {
      states.add(content);
    }
  }

  /**
   * Cases are tested with a level of ?x1
   * Baba is in the first position and tries to go left
   */
  void checkMoveSimple(
```

```
      @NotNull int[] content,
      boolean result,
      @NotNull int[][] possibleNextMoves) {
    LevelToTestTryToGo level = new LevelToTestTryToGo(
        content.length,
        1,
        content);
    State state = new State(level, content);
    assertEquals(
        result,
        state.tryToGo(0, Direction.RIGHT));
    assertEquals(possibleNextMoves.length, level.states.size());
    for (int i = 0; i < possibleNextMoves.length; i++) {
      assertArrayEquals(
          possibleNextMoves[i],
          level.states.get(i));
    }
  }

  @Test
  void testMoveEmpty() {
    // empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA,
            Tiles.EMPTY},
        false,
        new int[][]{new int[]{
            Tiles.EMPTY,
            Tiles.BABA}});
  }

  @Test
  void testMoveWall() {
    // wall
    checkMoveSimple(
        new int[]{
            Tiles.BABA,
            Tiles.WALL},
        false,
        new int[0][]);
  }

  @Test
  void testMoveFlag() {
    // flag
    checkMoveSimple(
        new int[]{
```

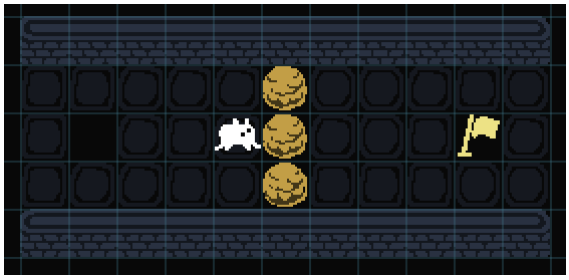
```
            Tiles.BABA,
            Tiles.FLAG},
        true,
        new int[0][]);
  }
}
```

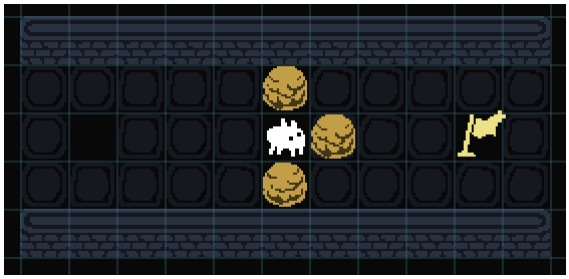
# It’s alive, and it can push rocks!

When moving a rock, the simplest case is to check if

- the rock is not against a border of the level
- there is nothing behind it

In this case Baba can go in this direction and push the rock.





*State.java*

```
case Tiles.ROCK:
  // reached the border of the level?
  if (!canGoThere(targetPosition, direction)) {
    return false;
  }
  // the position behind the rock
  int behindTheRockPosition = calculatePosition(targetPosition, direction);
  int behindTheRockPositionContent = content[behindTheRockPosition];
  // it it empty?
  if (behindTheRockPositionContent != Tiles.EMPTY) {
    return false;
  }
  // nice, build the new content
  newContent = content.clone();
  newContent[targetPosition] = Tiles.BABA;
  newContent[currentPosition] = Tiles.EMPTY;
  newContent[behindTheRockPosition] = Tiles.ROCK;
  level.addState(newContent);
  return false;
```

*StateTryToGoTest.java*

```
void testMoveRock() {
  // rock
  checkMoveSimple(
      new int[]{
          Tiles.BABA,
          Tiles.ROCK},
      false,
      new int[0][]);

  // rock | rock
  checkMoveSimple(
      new int[]{
          Tiles.BABA,
          Tiles.ROCK,
          Tiles.ROCK},
      false,
      new int[0][]);

  // rock | wall
  checkMoveSimple(
      new int[]{
          Tiles.BABA,
          Tiles.ROCK,
          Tiles.WALL},
      false,
      new int[0][]);

  // rock | empty
  checkMoveSimple(
      new int[]{
          Tiles.BABA,
          Tiles.ROCK,
          Tiles.EMPTY},
      false,
      new int[][]{new int[]{
          Tiles.EMPTY,
          Tiles.BABA,
          Tiles.ROCK}});
}
```

# It’s alive, it can push rocks, and not loop forever!

If I start the program now, it will run indefinitely. Not because of a bug but because of a missing feature: it lacks the ability to remember that a state has already been visited, thus doing the same thing again and again.

Java makes it a bit more complicated than it should do:

*Level.java*

```
/**
 * This set will be able to handle duplication of {@link State}
 * The custom {@link Comparator} is required to avoid
 * only comparing the arrays' addresses
 */
private final Set<int[]> pastStates =
  new TreeSet<>(new Comparator<>() {
    @Override
    public int compare(int[] o1, int[] o2) {
      for (int i = 0; i < size; i++) {
        int cmp = o2[i] - o1[i];
        if (cmp != 0) {
          return cmp;
        }
      }
      return 0;
    }
  });

void addState(@NotNull int[] content) {
  // only add if not already visited
  if (pastStates.add(content)) {
    states.add(new State(this, content));
  }
}
```

And… it works, the program can solve the level:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Solved in 00:00:00.001229643
```

\o/

In the next part I'll have a look on the solution that has been found.

# Part 4: printing a solution

Following the third part, where I solved the first level, this short part will deal with printing the solution.

## If the computer say it, it must be true!

I said I solved the first level, at least it's what the computer told me:

```
java -jar target/babaisyousolver-1.0-SNAPSHOT.jar levels/00
2019-05-08 23:03:06.827 babaisyousolver/levels/00 Reading level
2019-05-08 23:03:06.877 babaisyousolver/levels/00 Solving level
2019-05-08 23:03:06.882 babaisyousolver/levels/00 Solved in 00:00:00.001229643
```

But it would be nice if the code told me *how* it solved it, it would allow me to check if the solution looks reasonable, and it would allow me to use the solution to solve the level.

Which is the goal of the whole thing.

## States gonna be stateful

To implement this, a `State` must store the movements that lead to it, which means each `State` must concatenate the movement it adds to a list of previous movements.

*State.java*

```
final @NotNull byte[] previousMovements;

State(
    @NotNull Level level,
    @NotNull int[] content,
    @NotNull byte[] movements) {
  this.level = level;
  this.content = content;
  this.previousMovements = movements;
}

boolean tryToGo(
    int currentPosition,
    byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  int[] newContent;
  switch (targetPositionContent) {
    case Tiles.WALL:
      return false;
    case Tiles.EMPTY:
```

```
        newContent = content.clone();
        newContent[targetPosition] = Tiles.BABA;
        newContent[currentPosition] = Tiles.EMPTY;
        level.addState(newContent, addMovement(direction));
        return false;
      case Tiles.ROCK:
        // reached the border of the level?
        if (!canGoThere(targetPosition, direction)) {
          return false;
        }
        // the position behind the rock
        int behindTheRockPosition = calculatePosition(targetPosition, direction);
        int behindTheRockPositionContent = content[behindTheRockPosition];
        // it it empty?
        if (behindTheRockPositionContent != Tiles.EMPTY) {
          return false;
        }
        // nice, build the new content
        newContent = content.clone();
        newContent[targetPosition] = Tiles.BABA;
        newContent[currentPosition] = Tiles.EMPTY;
        newContent[behindTheRockPosition] = Tiles.ROCK;
        level.addState(newContent, addMovement(direction));
        return false;
      case Tiles.FLAG:
        return true;
      default:
        throw new IllegalArgumentException(
          Integer.toString(targetPositionContent));
    }
  }

  /**
   * Add a new movement at the end of the array
   */
  private @NotNull byte[] addMovement(byte movement) {
    int previousLength = previousMovements.length;
    byte[] result = new byte[previousLength + 1];
    System.arraycopy(previousMovements, 0, result, 0, previousLength);
    result[previousLength] = movement;
    return result;
  }
```

Then when the solution is found, I write it in the same directory as the level, so any change can be identified with a source control system. The file is deleted before trying to solve the level so I can detect when the solver fails.

Without pretty printing, the path would be something like →→→→→←←← which is not easy to read.

*App.java*

```
private static final String SOLUTION_FILES = "solution.txt";

private static void processLevel(
    @NotNull Path path
) throws IOException {
  print(path, "Reading level");
  LevelReader.LevelReaderResult levelReaderResult =
      LevelReader.readLevel(path);
  Level level = new Level(
      levelReaderResult.width,
      levelReaderResult.height,
      levelReaderResult.content);
  level.createInitStates();
  Path solutionFile = path.resolve(SOLUTION_FILES);
  Files.deleteIfExists(solutionFile);
  print(path, "Solving level");
  long startTime = System.nanoTime();
  State solution = level.solve();
  long stopTime = System.nanoTime();
  String endTime =
      LocalTime.MIN.plusNanos((stopTime - startTime)).
          toString();
  if (solution != null) {
    print(
        path,
        "Solved in " + endTime);
    writeSolution(solution.previousMovements, solutionFile);
  } else {
    print(
        path,
        "Failed in " + endTime);
  }
}

private static void writeSolution(
    @NotNull byte[] solution,
    @NotNull Path solutionPath)
    throws IOException {
  List<String> steps = new ArrayList<>();
  byte currentMovement = -1;
  int numberOfMovesThisWay = 1;
  for (byte c : solution) {
    if (c != currentMovement) {
      if (currentMovement != -1) {
        steps.
            add(
```

```
                Integer.toString(numberOfMovesThisWay) +
                Direction.VISUAL[currentMovement]);
      }
      currentMovement = c;
      numberOfMovesThisWay = 1;
    } else {
      numberOfMovesThisWay += 1;
    }
  }
  // complete with last move
    steps.
        add(
            Integer.toString(numberOfMovesThisWay) +
            Direction.VISUAL[currentMovement]);

  String content = String.join(" ", steps);
  Files.write(solutionPath, content.getBytes());
}
```

The result:

*levels/00/solution.txt*

```
6→ 3← 1↓ 5→
```

# Wait, what?

Huh? The last step is missing?!

Which is normal, since the data used is the path to the last `State`, which lacks the last movement done inside the `State`.

The solution is to add the last movement to the past movements, and to return this value as the result, it requires a bunch of modifications but the migration is easy.

*State.java*

```
/**
 * Process the current state
 * @return a list of {@link Direction} if a solution is found,
 * else null
 */
@Nullable byte[] process() {
  int babaPosition = findBaba();
  int babaLine = babaPosition / level.width;
  int babaColumn = babaPosition % level.width;

  byte[] result;
  // Up
```

```
  if (babaLine > 0) {
    result = tryToGo(babaPosition, Direction.UP);
    if (result != null) {
      return result;
    }
  }
  // same for other directions
  return null;
}

/**
 * Try to go on a direction from a position
 * @return a list of {@link Direction} if a solution is found,
 * else null
 */
@Nullable byte[] tryToGo(
    int currentPosition,
    byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  int[] newContent;
  switch (targetPositionContent) {
    case Tiles.WALL:
      return null;
    case Tiles.EMPTY:
      newContent = content.clone();
      newContent[targetPosition] = Tiles.BABA;
      newContent[currentPosition] = Tiles.EMPTY;
      level.addState(newContent, addMovement(direction));
      return null;
    case Tiles.ROCK:
      // reached the border of the level?
      if (!canGoThere(targetPosition, direction)) {
        return null;
      }
      // the position behind the rock
      int behindTheRockPosition = calculatePosition(targetPosition, direction);
      int behindTheRockPositionContent = content[behindTheRockPosition];
      // it it empty?
      if (behindTheRockPositionContent != Tiles.EMPTY) {
        return null;
      }
      // nice, build the new content
      newContent = content.clone();
      newContent[targetPosition] = Tiles.BABA;
      newContent[currentPosition] = Tiles.EMPTY;
      newContent[behindTheRockPosition] = Tiles.ROCK;
```

```
        level.addState(newContent, addMovement(direction));
        return null;
      case Tiles.FLAG:
        return addMovement(direction);
      default:
        throw new IllegalArgumentException(
          Integer.toString(targetPositionContent));
    }
  }
```

In `Level` the value is propagated.

*Level.java*

```
@Nullable byte[] solve() {
    while (true) {
      State state = states.pop();
      if (state == null) {
        return null;
      }
      byte[] result = state.process();
      if (result != null) {
        return result;
      }
    }
  }
```

And in `App`, this value is printed.

*App.java*

```
byte[] solution = level.solve();
long stopTime = System.nanoTime();
String endTime =
  LocalTime.MIN.plusNanos((stopTime - startTime)).
      toString();
if (solution != null) {
  print(
    path,
    "Solved in " + endTime);
  writeSolution(solution, solutionFile);
```

Which prints the right solution

*levels/00/solution.txt*

```
6→ 3← 1↓ 5→ 1↑
```

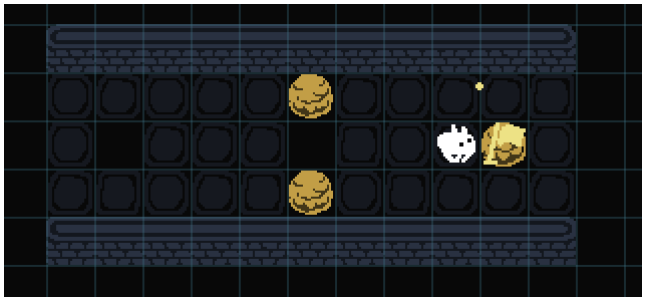
I'm skipping the refactoring to do in the tests here since they are simple and don't bring anything.

In the next part I'll add some missing behaviors.

# Part 5: adding missing behaviors

Following the fourth part, where I coded what was missing to display a solution, I'll now improve the behavior of the solver to make it closer to the game.

## Two elements on one cell



*2 elements 1 cell*

The first behavior to change is the ability to have two elements on the same cell.

It conflicts with the way data are organized in `State` where one `int` is used for each cell, representing the id of the corresponding element.

Instead of an `int` I could use an array of `int` for cell, `State#content` would then become an `int[][]` but by experience I try to avoid this situation since it would require adding array iterations in many places and lots of allocations which increase computation time, for example when checking if a position has already been reached.

Several elements can be on one cell, but at most one element of each type. In this case using bit fields is a common pattern. It's a bit like an array of boolean of a fixed size packed as a primitive value.

Each kind of element will be represented by an index, and the presence of a bit with an index indicates that the corresponding element is on the cell.

Example, if 8 bit are used to store the data, rock at index 3 and flag on index 5:

```
76543210 bit index
00000000 empty cell
00001000 cell with a rock
00100000 cell with a flag
00101000 cell with a rock and a flag
```

In Java `int` use 32 bits, so it's enough to store 32 types of elements. It's enough for now, and I'll be able to come back to it later if it's required.

This means `State#content` will still be an `int[]`, so signatures won't change but the code will have to be updated.

The bit manipulation syntax is described here.

Represening an empty cell with the `0` (aka `00000000 00000000 00000000 00000000`) value would be nice because it makes testing for an empty cell easier, so the first step is to modify the

declaration order in `Tiles` to match this idea.

*Tiles.java*

```
interface Tiles {

  String EMPTY_STRING = "empty";
  String BABA_STRING = "Baba";
  // …

  String[] ALL_STRINGS = new String[]{
      EMPTY_STRING,
      BABA_STRING,
      // …
  };

  int EMPTY = 0;
  int BABA = EMPTY + 1;
  // …

}
```

Then I add bit masks for the elements that will be used to detect and modify content of the associated type.

*Tiles.java*

```
int BABA_MASK = 1 << (BABA - 1);
int BABA_TEXT_MASK = 1 << (BABA_TEXT - 1);
int FLAG_MASK = 1 << (FLAG - 1);
int FLAG_TEXT_MASK = 1 << (FLAG_TEXT - 1);
int IS_TEXT_MASK = 1 << (IS_TEXT - 1);
int PUSH_TEXT_MASK = 1 << (PUSH_TEXT - 1);
int ROCK_MASK = 1 << (ROCK - 1);
int ROCK_TEXT_MASK = 1 << (ROCK_TEXT - 1);
int STOP_TEXT_MASK = 1 << (STOP_TEXT - 1);
int WALL_MASK = 1 << (WALL - 1);
int WALL_TEXT_MASK = 1 << (WALL_TEXT - 1);
int WIN_TEXT_MASK = 1 << (WIN_TEXT - 1);
int YOU_TEXT_MASK = 1 << (YOU_TEXT - 1);
```

A bit mask are used this way:

```
if ((value & BABA_MASK) != Tiles.EMPTY) // check if the cell contains Baba
value = (value | BABA_MASK); // add Baba to the cell
value = (value ^ BABA_MASK); // remove Baba from the cell
```

As the empty element is equal to `0` there is no associated bit mask, an empty cell has a content

equals to `0`.

Since the meaning of `State#content` is changed, the first step is `Level#createInitStates` where the first `State` is created. Instead of using the value of the element the bit at the right position is set.

*Level.java*

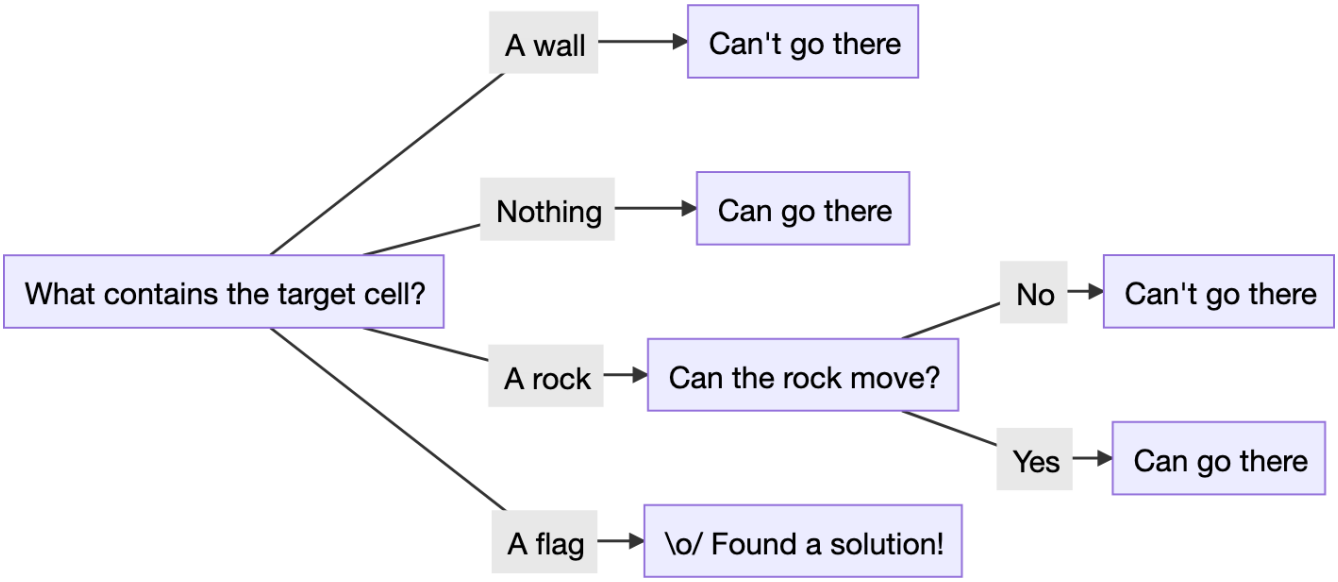
```
void createInitStates() {
int[] contentForState = new int[size];
for (int i = 0; i < size; i++) {
    int originalTile = originalContent[i];
    if (originalTile != 0) {
    // the bit mask is derived from the tile index
    contentForState[i] = (1 << originalTile - 1);
    }
}
addState(contentForState, new byte[0]);
}
```

Next finding Baba use the bit mask:

*State.java*

```
/**
 * Find the index of the baba position.
 *
 * @return the position or -1 if not found
 */
private int findBaba() {
    for (int i = 0; i < level.size; i++) {
      if ((content[i] & Tiles.BABA_MASK) != 0) {
        return i;
      }
    }
    return -1;
}
```

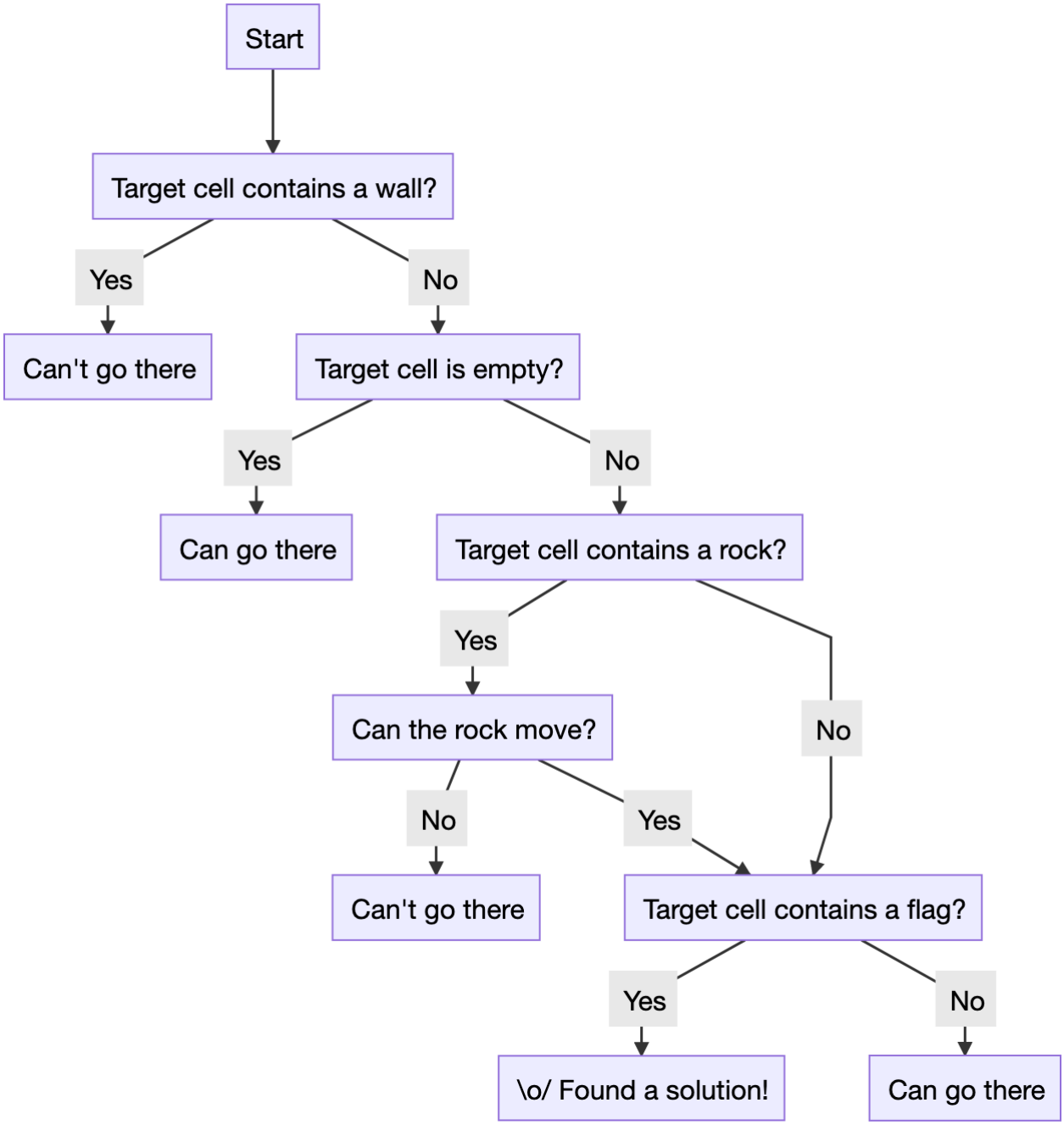
Then the *pièce de résistance*: `State#tryToGo`. Here it's not only about migrating the data representation but also the behavior.




*The existing algo*

The existing code was a `switch`, since all cases were independent from each others.

Dealing with the new case add a depencency between the rock case and the flag case:




*The new algo*

This ensure a flag can't be reached if it's on the same cell that a rock that can't be moved.

So the code must be rewritten as several `if`.

*State.java*

```
/**
 * Try to go on a direction from a position
 * @return a list of {@link Direction} if a solution is found,
 * else null
 */
@Nullable byte[] tryToGo(
    int currentPosition,
    byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
```

```
int targetPositionContent = content[targetPosition];

// target contains a wall
if ((targetPositionContent & Tiles.WALL_MASK) != Tiles.EMPTY) {
  return null;
}
int[] newContent = content.clone();

// target is empty
if (targetPositionContent == Tiles.EMPTY) {
  newContent[targetPosition] |= Tiles.BABA_MASK;
  newContent[currentPosition] ^= Tiles.BABA_MASK;
  level.addState(newContent, addMovement(direction));
  return null;
}

if ((targetPositionContent & Tiles.ROCK_MASK) != Tiles.EMPTY) {
  // reached the border of the level?
  if (!canGoThere(targetPosition, direction)) {
    return null;
  }
  // the position behind the rock
  int behindTheRockPosition =
      calculatePosition(targetPosition, direction);
  int behindTheRockPositionContent =
      content[behindTheRockPosition];

  // does it block the move?
  int blockingMask = Tiles.ROCK_MASK | Tiles.WALL_MASK;
  if ((behindTheRockPositionContent & blockingMask) != Tiles.EMPTY) {
    return null;
  }

  // nice, build the new content
  // remove the rock
  targetPositionContent = newContent[targetPosition] ^ Tiles.ROCK_MASK;
  newContent[targetPosition] = targetPositionContent;
  // add a rock at the end
  newContent[behindTheRockPosition] =
      behindTheRockPositionContent | Tiles.ROCK_MASK;
}

if ((targetPositionContent & Tiles.FLAG_MASK) != Tiles.EMPTY) {
  return addMovement(direction);
}

// move Baba
newContent[targetPosition] |= Tiles.BABA_MASK;
```

```
    newContent[currentPosition] ^= Tiles.BABA_MASK;
    level.addState(newContent, addMovement(direction));
    return null;
}
```

Now it's time to migrate existing tests and to add new ones to ensure I didn't miss any case:

*StateTryToGoTest.java*

```
@Test
void testMoveEmpty() {
  // empty
  checkMoveSimple(
      new int[]{
          Tiles.BABA_MASK,
          Tiles.EMPTY},
      null,
      new int[][]{new int[]{
          Tiles.EMPTY,
          Tiles.BABA_MASK}});
}

@Test
void testMoveWall() {
  // wall
  checkMoveSimple(
      new int[]{
          Tiles.BABA_MASK,
          Tiles.WALL_MASK},
      null,
      new int[0][]);
}

@Test
void testMoveFlag() {
  // flag
  checkMoveSimple(
      new int[]{
          Tiles.BABA_MASK,
          Tiles.FLAG_MASK},
      new byte[]{Direction.RIGHT},
      new int[0][]);
}

@Test
void testMoveRock() {
  // rock
  checkMoveSimple(
```

```
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK},
        null,
        new int[0][]);

    // rock + flag
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK | Tiles.FLAG_MASK},
        null,
        new int[0][]);

    // rock | rock
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.ROCK_MASK},
        null,
        new int[0][]);

    // rock | wall
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.WALL_MASK},
        null,
        new int[0][]);

    // rock | empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.EMPTY},
        null,
        new int[][]{new int[]{
            Tiles.EMPTY,
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK}});

    // rock + flag | empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
```

```
            Tiles.ROCK_MASK | Tiles.FLAG_MASK,
            Tiles.EMPTY},
        new byte[]{Direction.RIGHT},
        new int[0][]);

    // rock | flag
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.FLAG_MASK},
        null,
        new int[][]{new int[]{
            Tiles.EMPTY,
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK | Tiles.FLAG_MASK}});

    // rock | rock | empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.ROCK_MASK,
            Tiles.EMPTY},
        null,
        new int[][]{new int[]{
            Tiles.EMPTY,
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.ROCK_MASK}});

    // rock | rock | wall | empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK,
            Tiles.ROCK_MASK,
            Tiles.WALL_MASK,
            Tiles.EMPTY},
        null,
        new int[0][]);

    // rock + flag | rock | empty
    checkMoveSimple(
        new int[]{
            Tiles.BABA_MASK,
            Tiles.ROCK_MASK | Tiles.FLAG_MASK,
            Tiles.ROCK_MASK,
```

```
            Tiles.EMPTY},
        new byte[]{Direction.RIGHT},
        new int[0][]);
}
```

The code used to check if a position has already been reached don't need to be updated since it relies on the values of the content: the inner values changed but comparing them is still the right thing to do.

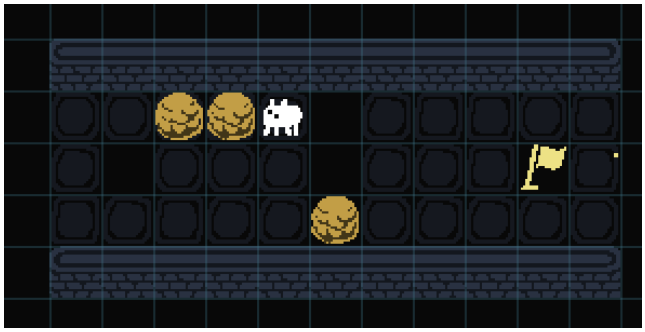
Running the program show the new capability being used, making the solution shorter:

*levels/00/solution.txt*

```
8→
```

# Pushing several rocks

The second behavior to change is the ability to push several rock at once.



*Here comes the rockipede*

When baba tries to move a rock, the code need to iterate until it finds either:

- something it can push the rocks on (an empty cell or a cell with a flag)
- a wall
- the border of the level

This time there is no signature change but only some code to add, which means it's easier to start with the tests:

*StateTryToGoTest.java*

```
// rock | rock | empty
checkMoveSimple(
    new int[]{
        Tiles.BABA_MASK,
        Tiles.ROCK_MASK,
        Tiles.ROCK_MASK,
        Tiles.EMPTY},
    null,
    new int[][]{new int[]{
        Tiles.EMPTY,
        Tiles.BABA_MASK,
        Tiles.ROCK_MASK,
        Tiles.ROCK_MASK}});

// rock | rock | wall | empty
checkMoveSimple(
    new int[]{
        Tiles.BABA_MASK,
        Tiles.ROCK_MASK,
        Tiles.ROCK_MASK,
        Tiles.WALL_MASK,
        Tiles.EMPTY},
    null,
    new int[0][]);

// rock + flag | rock | empty
checkMoveSimple(
    new int[]{
        Tiles.BABA_MASK,
        Tiles.ROCK_MASK | Tiles.FLAG_MASK,
        Tiles.ROCK_MASK,
        Tiles.EMPTY},
    new byte[]{Direction.RIGHT},
    new int[0][]);
```

Then the code:

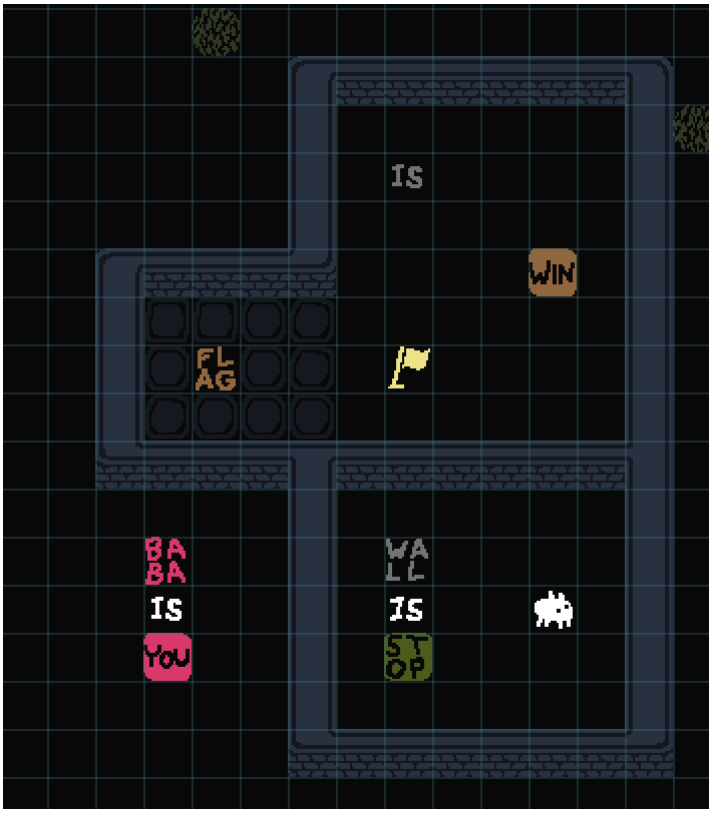
*State.java*

```
if ((targetPositionContent & Tiles.ROCK_MASK) != Tiles.EMPTY) {
  boolean foundCellAfterRocks = false;
  int candidatePosition = targetPosition;
  // explore the next cells until the right stop,
  // a wall or the end of the level
  while (!foundCellAfterRocks) {
    // reached the border of the level?
    if (!canGoThere(candidatePosition, direction)) {
      return null;
    }
    // the position behind the current position
    int behindCandidatePosition =
        calculatePosition(candidatePosition, direction);
    int behindCandidatePositionContent =
        content[behindCandidatePosition];

    // is it a wall?
    if ((behindCandidatePositionContent & Tiles.WALL_MASK) != Tiles.EMPTY) {
      return null;
    }

    // is it another rock?
    if ((behindCandidatePositionContent & Tiles.ROCK_MASK) != Tiles.EMPTY) {
      // yes another rock, next step of the loop
      candidatePosition = behindCandidatePosition;
    } else {
      // no rock, found a solution!
      foundCellAfterRocks = true;
      // build the new content
      // remove the rock near Baba
      targetPositionContent =
          newContent[targetPosition] ^ Tiles.ROCK_MASK;
      newContent[targetPosition] = targetPositionContent;
      // add a rock at the end
      newContent[behindCandidatePosition] =
          behindCandidatePositionContent | Tiles.ROCK_MASK;
    }
  }
}
```

Now all visible behaviors of the first level are implemented. In the next part I'll start working on the second level, where it will be possible to change the rules.

# Part 6: changing the rules

Following the fifth part, where I coded missing behaviors, I'll work on the second level on being able to modify the rules.




*Hello level two my old friend*

I'll follow the same steps than for the first level which means I'll first implement the minimal subset of features required to find a solution.

## First: thinking

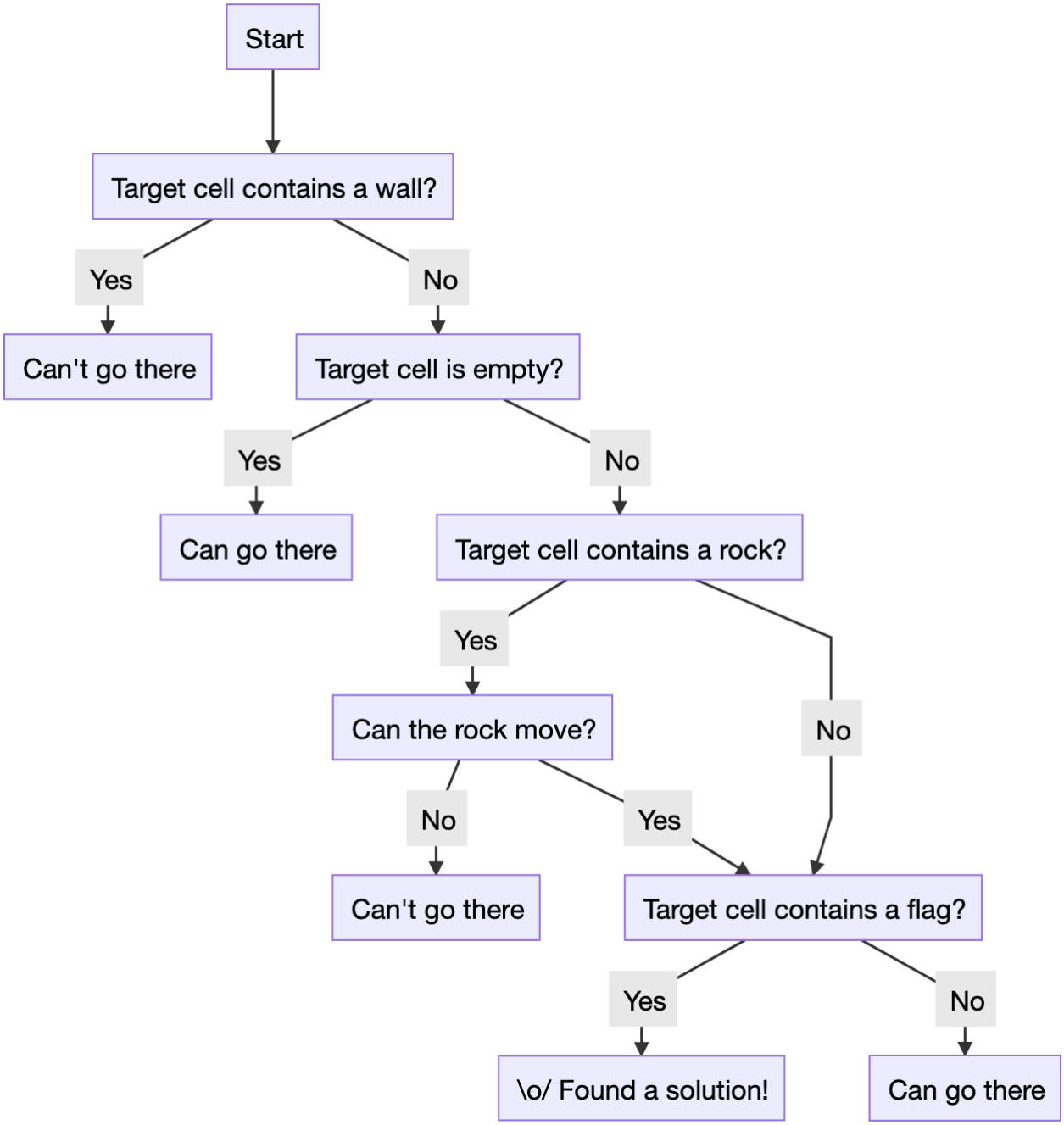
Before starting the implementation I'll need to plan ahead how to deal with the changing rules.

Looking at the level, the rules are establishing capabilities about game elements: this thing can do this thing, that other thing can do that thing.

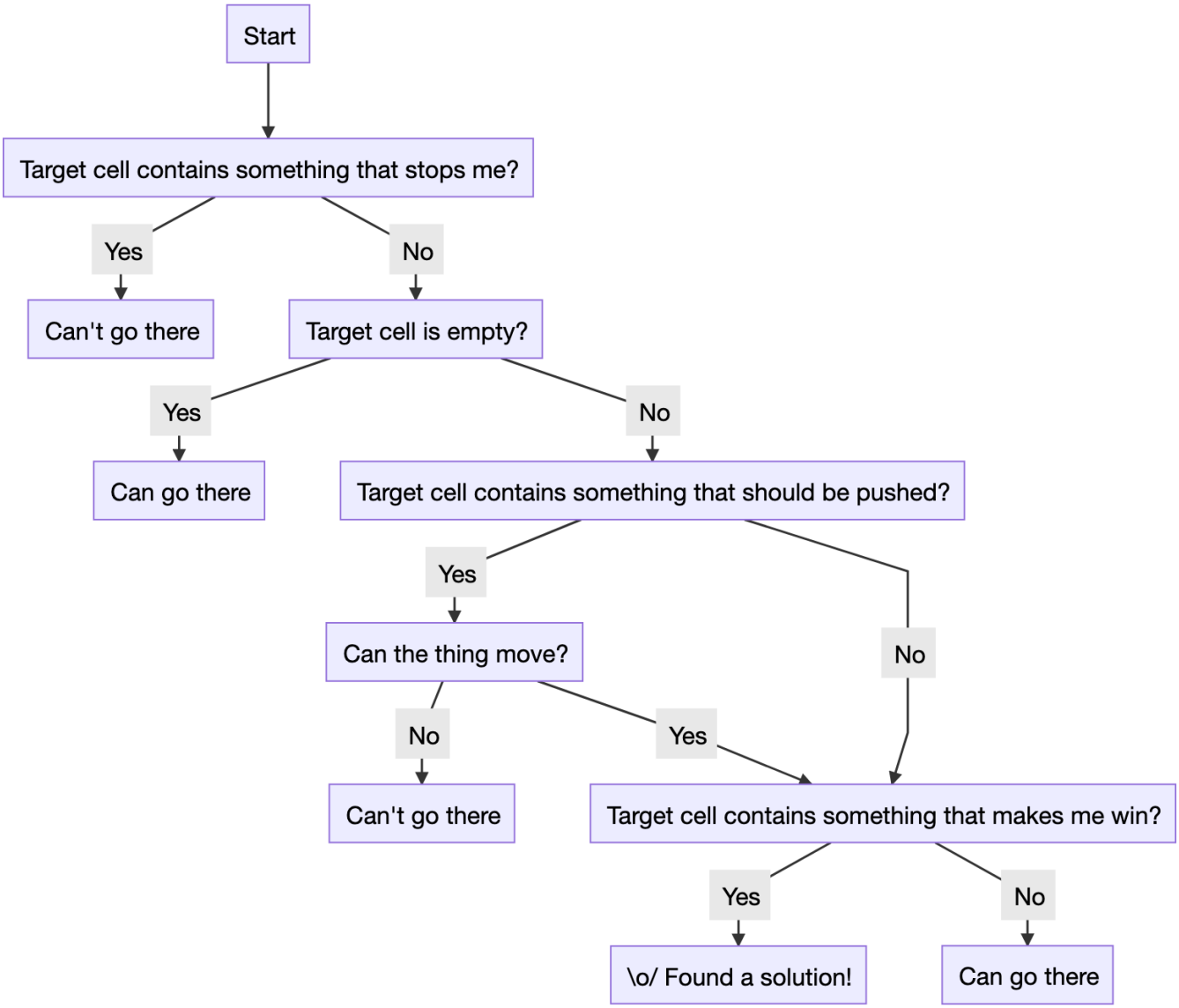
By default you can walk through most elements except the tiles that represents rules components.

So I'll try scanning for the rules, adding capabilities as I find them, then process the movements.

Updating the movement code will rely on generalizing the existing behaviors. For example instead of checking if Baba is stopped by walking against a wall, I'll check if Baba is walking against something that has the stop capability.


Start
Target cell contains a wall?
Yes
No
Can't go there
Target cell is empty?
Yes
No
Can go there
Target cell contains a rock?
Yes
No
Can the rock move?
No
Yes
Can't go there
Target cell contains a flag?
Yes
No
\o/ Found a solution!
Can go there

*From this*




*To this*

To store the capabilities I'll use bit fields again. For each capability I'll use a bit field to store which kind of elements the capability should be applied to, using the same mask value used to define which elements are on each tile.

For example

```
if ((targetPositionContent & Tiles.ROCK_MASK) != Tiles.EMPTY) {
```

Will become

```
if ((targetPositionContent & stopTilesMask) != Tiles.EMPTY) {
```

To scan for the rules I need to be able to identify the corresponding tiles, from what I've seen from the game so far, there are three kind of rule tiles:

- the `IS` tile
- the subject tiles, like `Baba` or `flag`
- the tiles that define a capability, like `stop` or `push`

A rule—as a first approximation—use the `subject IS definition` form, with the three tiles placed horizontally or vertically.

## Second: generating

So I need to add some info to the `Tile` class to define which tiles are subjects, which tiles are definitions and so on.

I don't want to write this code manually since it would be tedious and error-prone but would prefer to declare it and let the machine deal with it.

So times to do some code generation, which is less scary that it sounds.

Except that the first step is to deal with Maven to transform our single-module project into a multi-modules project, which is definitely scary (and tedious and error-prone).

I skip the boring part, you can find the result in the source, but the target structure is:

```
pom.xml ⑥
|__engine ⑤
|  |__pom.xml
|  |__…
|__levels ⑦
|  |__00
|  |  |__content.txt
|  |  |__solution.txt
|  |  |__tiles.txt
|  |__…
|__tiles ③
|  |__pom.xml
|  |__src
|     |__main
|       |__resources
|         |__tiles.json ④
|__tiles-generator ①
   |__pom.xml
   |__src
      |__main
        |__java
          |__net
            |__archiloque
              |__babaisyousolver
                |__tilesgenerator
                  |__TilesGenerator.java ②
```

① Module containing the code generation routine

② Class doing the code generation

③ Module containing the tiles declaration, during build it will use `TilesGenerator` to transform it into a Java file

④ JSON file where the data are declared

⑤ Module containing the solver engine, that will use the generated `Tile` class so depends of the `tiles` module

⑥ Parent project that declare all the submodules

⑦ Game levels

I use JSON for declaring the data because they are not complex enough to delve in XML, and because I hate YAML with the burning passion of a thousand suns.

I'll implement the generation in two steps: first I'll generate an equivalent of the existing file, then I'll add the new features.

The first version of the JSON file:

*tiles.json*

```
{
  "empty": {},
  "Baba": {},
  "Baba text": {},
  "flag": {},
  "flag text": {},
  "is text": {},
  "push text": {},
  "rock": {},
  "rock text": {},
  "stop text": {},
  "wall": {},
  "wall text": {},
  "win text": {},
  "you text": {}
}
```

I use JavaPoet to generate the code instead of doing it manually.

The resulting code is a bit verbose because of the builder pattern used everywhere, but it's easy to write and read.

*TilesGenerator.java*

```
/**
 * Entry point to generate the tiles class
 */
public class TilesGenerator {

  private static final String TILES_JSON_FILES = "tiles.json";

  public static void main(String[] args)
      throws Exception {
    if (args.length != 1) {
```

```
      throw new IllegalArgumentException(
          "Need one argument and got " + args.length);
    }
    String targetDir = args[0];
    System.out.println("Will generate in [" + targetDir + "]");

    // get the JSON content
    URI jsonResourceUri = TilesGenerator.class.
        getClassLoader().
        getResource(TILES_JSON_FILES).
        toURI();
    String tilesFileContent = Files.
        readString(Path.of(jsonResourceUri));
    JSONObject jsonObject = new JSONObject(tilesFileContent);
    System.out.println(jsonObject.length() + " tiles found");
    new TilesGenerator(jsonObject).
        generate().
        writeTo(new File(targetDir));
  }

  private final @NotNull JSONObject jsonObject;

  private final @NotNull TypeSpec.Builder tileInterface =
      TypeSpec.
          interfaceBuilder("Tiles");

  private final @NotNull List<String> fields =
      new ArrayList<>();

  private TilesGenerator(
      @NotNull JSONObject jsonObject) {
    this.jsonObject = jsonObject;
  }

  private @NotNull JavaFile generate() {
    // annotation to indicates the class is generated
    AnnotationSpec generatedAnnotation =
        AnnotationSpec.builder(
            Generated.class).
            addMember(
                "value",
                "$S",
                TilesGenerator.class.getName()).
            build();

    // initialize the interface
    tileInterface.
        addModifiers(Modifier.PUBLIC).
```

```
        addAnnotation(
            generatedAnnotation);

    // create a list with all values
    // being sure empty is first
    for (String fieldName : jsonObject.keySet()) {
      if (!fieldName.equals("empty")) {
        fields.add(fieldName);
      }
    }
    fields.sort(String::compareToIgnoreCase);
    fields.add(0, "empty");


    List<String> fieldsNames = new ArrayList<>();

    // declare the XXX_STRING String constants
    for (String field : fields) {
      // create the constant from the field name
      String fieldConstantName =
          field.
              toUpperCase().
              replace(' ', '_');
      fieldsNames.add(fieldConstantName);
      addField(
          String.class,
          fieldConstantName + "_STRING",
          "$S", field
      );
    }

    // declare the ALL_STRINGS containing the XXX_STRING constants
    CodeBlock.Builder allStringCode = CodeBlock.
        builder().
        add(" new String[]{\n");
    for (String fieldName : fieldsNames) {
      allStringCode.add(fieldName + "_STRING,\n");
    }
    allStringCode.
        add("}");

    addField(
        ArrayTypeName.of(String.class),
        "ALL_STRINGS",
        allStringCode.build()
    );

    // the int values
```

```
    for (int i = 0; i < fieldsNames.size(); i++) {
      addField(
          TypeName.INT,
          fieldsNames.get(i),
          Integer.toString(i));
    }

    // the masks  (no masks for empty)
    for (int i = 1; i < fieldsNames.size(); i++) {
      addField(
          TypeName.INT,
          fieldsNames.get(i) + "_MASK",
          Integer.toString(1 << +(i - 1)));
    }

    // create the file
    return JavaFile.builder(
        "net.archiloque.babaisyousolver",
        tileInterface.build()).
        build();
  }

  /**
   * Create a field from parameters
   */
  private void addField(
      @NotNull TypeName typeName,
      @NotNull String fieldName,
      @NotNull String content) {
    tileInterface.addField(
        FieldSpec.
            builder(
                typeName,
                fieldName,
                Modifier.PUBLIC,
                Modifier.STATIC,
                Modifier.FINAL).
            initializer(content).
            build());
  }

  /**
   * Create a field from parameters
   */
  private void addField(
      @NotNull TypeName typeName,
      @NotNull String fieldName,
      @NotNull CodeBlock codeBlock) {
```

```
    tileInterface.addField(
        FieldSpec.
            builder(
                typeName,
                fieldName,
                Modifier.PUBLIC,
                Modifier.STATIC,
                Modifier.FINAL).
            initializer(codeBlock).
            build());
  }

  /**
   * Create a field from parameters
   */
  private void addField(
      @NotNull Class type,
      @NotNull String fieldName,
      @NotNull String format,
      Object... args) {
    tileInterface.addField(
        FieldSpec.
            builder(
                type,
                fieldName,
                Modifier.PUBLIC,
                Modifier.STATIC,
                Modifier.FINAL).
            initializer(format, args).
            build());
  }

}
```

Then I can run the tests and the app and checks nothing is broken.

I then add the information related to the tiles.

*tiles.json*

```
{
  "empty": {},
  "Baba": {},
  "Baba text": {
    "text": true,
    "subject": true,
    "subjectTarget": "Baba"
  },
  "flag": {},
  "flag text": {
    "text": true,
    "subject": true,
    "subjectTarget": "flag"
  },
  "is text": {
    "text": true
  },
  "push text": {
    "text": true,
    "definition": true
  },
  "rock": {},
  "rock text": {
    "text": true,
    "subject": true,
    "subjectTarget": "rock"
  },
  "stop text": {
    "text": true,
    "definition": true
  },
  "wall": {},
  "wall text": {
    "text": true,
    "subject": true,
    "subjectTarget": "wall"
  },
  "win text": {
    "text": true,
    "definition": true
  },
  "you text": {
    "text": true,
    "definition": true
  }
}
```

Each subject contains a link to the entity it deals with so the code can find it. Since the naming is consistent (the text is named from the entity with " text" appended) I could deduce the entity automatically, but it would be a bit too magical for my taste.

The resulting data in the `Tiles`:

*Tiles.java*

```
int TEXT_MASKS = 7610;

int SUBJECT_MASKS = 1162;

int DEFINITION_MASKS = 6432;

static int getTarget(int sourceMask) {
  switch (sourceMask) {
  case Tiles.WALL_TEXT_MASK:
    return WALL_MASK;
  case Tiles.BABA_TEXT_MASK:
    return BABA_MASK;
  case Tiles.FLAG_TEXT_MASK:
    return FLAG_MASK;
  case Tiles.ROCK_TEXT_MASK:
    return ROCK_MASK;
  default:
    throw new IllegalArgumentException(Integer.toString(sourceMask));
  }
}
```

The link between the text and the entities is done through a switch method instead of an array because the lookup key is not the text id but the text mask. For example `ROCK_TEXT_MASK` is equal to 128, which means an array of 128 elements with only four useful values.

I could change the tiles' order to put these ones first and keep the array short, but I like the alphabetical order that is dislike the switch.

# Third: coding

Now the data are available I can work on the new behaviors.

First I add new members to `State` to contain the capabilities.

*State.java*

```
int pushTilesMask = Tiles.TEXT_MASKS;

int stopTilesMask = Tiles.EMPTY;

int youTilesMask = Tiles.EMPTY;

int winTilesMask = Tiles.EMPTY;
```

The text files are pushable by default, the other capabilities are empty at the beginning.

Then the code, which is a direct translation of the algorithm I described before. I first look for `IS` tiles, then look if the statement if well formed, then add the capability.

*State.java*

```
void processRules() {
  // locate the "IS"
  for (int i = 0; i < level.size; i++) {
    if ((content[i] & Tiles.IS_TEXT_MASK) != Tiles.EMPTY) {
      // any room to make an horizontal sentence ?
      int isLine = i / level.width;
      if ((isLine > 0) && (isLine < (level.height - 1))) {
        checkRule(
            i - level.width,
            i + level.width);
      }

      // any room to make a vertical sentence ?
      int isColumn = i % level.width;
      if ((isColumn > 0) && (isColumn < (level.width - 1))) {
        checkRule(
            i - 1,
            i + 1);
      }
    }
  }
}

private void checkRule(
    int beforeCellIndex,
    int afterCellIndex) {
  // validate it's a rule
  int subject = content[beforeCellIndex] &
      Tiles.SUBJECT_MASKS;
  if (subject == Tiles.EMPTY) {
    return;
  }
```

```
    int definition = content[afterCellIndex] &
        Tiles.DEFINITION_MASKS;
    if (definition == Tiles.EMPTY) {
      return;
    }

    // apply the result
    int targetMask = Tiles.TARGET_MASKS[subject];
    switch (definition) {
      case Tiles.PUSH_TEXT_MASK:
        pushTilesMask |= targetMask;
        return;
      case Tiles.STOP_TEXT_MASK:
        stopTilesMask |= targetMask;
        return;
      case Tiles.WIN_TEXT_MASK:
        winTilesMask |= targetMask;
        return;
      case Tiles.YOU_TEXT_MASK:
        youTilesMask |= targetMask;
        return;
      default:
        throw new IllegalArgumentException(Integer.toString(definition));
    }
  }
```

The “YOU” behavior is more complex to implement because it requires to add new business logic, whereas the other only need small code changes.

So I’ll implement the other rules first so I’ll be able to find a solution for the second level and I’ll deal with it later.

I won’t change all the behaviors at once and I’ll skip the `YOU` part first.

Nearly all the changes are one-liners, were the hardcoded tiles like `Tiles#ROCK_MASK` are replaced by the “behavior-masks” like `stopTilesMask`.

The only exception is the pushing part.

When Baba was only pushing rocks, the only thing to do was to teleport the first rock at the end.

From

```
brrr
```

To

```
 brrr
```

But now all the elements must be moved one by one as the algo looks for a suitable cell.

*State.java*

```
/**
 * Try to go on a direction from a position
 *
 * @return a list of {@link Direction} if a solution is found,
 * else null
 */
@Nullable byte[] tryToGo(
    int currentPosition,
    byte direction) {
  int targetPosition = calculatePosition(currentPosition, direction);
  int targetPositionContent = content[targetPosition];

  // target contains something that stops me
  if ((targetPositionContent & stopTilesMask) != Tiles.EMPTY) {
    return null;
  }
  int[] newContent = content.clone();

  // target is empty
  if (targetPositionContent == Tiles.EMPTY) {
    newContent[targetPosition] |= Tiles.BABA_MASK;
    newContent[currentPosition] ^= Tiles.BABA_MASK;
    level.addState(newContent, addMovement(direction));
    return null;
  }

  int currentPushingMask = targetPositionContent & pushTilesMask;
  if (currentPushingMask != Tiles.EMPTY) {
    // remove the pushed elements
    targetPositionContent &= (~pushTilesMask);

    int candidatePosition = targetPosition;
    // explore the next cells until the right stop,
    // a wall or the end of the level
    while (currentPushingMask != Tiles.EMPTY) {
      // reached the border of the level?
      if (!canGoThere(candidatePosition, direction)) {
        return null;
      }
      // the position behind the current position
      int behindCandidatePosition =
          calculatePosition(candidatePosition, direction);
      int behindCandidatePositionContent =
          newContent[behindCandidatePosition];

      // is it something that stop me
      if ((behindCandidatePositionContent & stopTilesMask) != Tiles.EMPTY) {
        return null;
```

```
      }

      // is it another thing that should be pushed?
      int behindCandidatePushingMask = behindCandidatePositionContent & pushTilesMask;
      if ((behindCandidatePushingMask) != Tiles.EMPTY) {
        // yes another thing to push

        // remove the pushed thing from next cell
        // and add the thing that was being pushed
        behindCandidatePositionContent =
            behindCandidatePositionContent &
                (~behindCandidatePushingMask) |
                currentPushingMask;
        newContent[behindCandidatePosition] = behindCandidatePositionContent;
        currentPushingMask = behindCandidatePushingMask;
        candidatePosition = behindCandidatePosition;
      } else {
        // found a cell that suits us!
        // add the thing that was being pushed

        // build the new content
        newContent[targetPosition] = targetPositionContent;
        // add a rock at the end
        newContent[behindCandidatePosition] |= currentPushingMask;
        currentPushingMask = Tiles.EMPTY;
      }
    }
  }

  if ((targetPositionContent & winTilesMask) != Tiles.EMPTY) {
    return addMovement(direction);
  }

  // move Baba
  newContent[targetPosition] |= Tiles.BABA_MASK;
  newContent[currentPosition] ^= Tiles.BABA_MASK;
  level.addState(newContent, addMovement(direction));
  return null;
}
```

# Fourth: solving

After checking that the first level still works, it's time to have a look at the second one :

*levels/01/content.txt*

```
    wwwwwwww
    w      w
    w I    w
    w      w
wwwww    ! w
w           w
w F   f     w
w           w
wwwwwwwwwwww
    w      w
 B  w W    w
 I  w I  b w
 Y  w S    w
    w      w
    wwwwwwww
```

And … it works, in a kind of way at least:

*levels/01/solution.txt*

```
1→ 2↓ 5← 1↑ 5→ 9← 9→ 1↓ 1→ 1↑ 10← 1↓ 1← 1↑ 11→ 1↑ 5← 5→ 2↓ 11← 2↑ 9→ 9← 2↓ 11→ 2↑ 1← 1→ 2↓ 11←
2↑ 10→ 10← 2↓ 11→ 2↑ 11← 1↑ 8→ 8← 2↓ 11→ 2↑ 1← 1→ 2↓ 11← 2↑ 9→ 9← 2↓ 11→ 2↑ 1← 1→ 2↓ 11← 2↑ 10→
10← 2↓ 11→ 2↑ 11← 1↑ 10→ 1↓ 1→ 1↑ 11← 1↑ 10→ 1↓ 1→ 1↑ 11← 1↑ 10→ 1↓ 1→ 1↑ 11← 1↑ 9→ 9← 2↓ 11→ 2↑
1← 1→ 2↓ 11← 2↑ 10→ 10← 2↓ 11→ 2↑ 11← 1↑ 10→ 1↑ 1← 1→ 2↓ 1→ 2↓ 11← 2↑ 8→ 2↑ 2→ 2↓ 1→ 2↓ 11← 2↑
6→ 2↑ 5← 1↑ 1← 1↓ 10→ 2↑ 4← 4→ 2↓ 10← 1↓ 10→ 1↓ 1→ 2↓ 11← 3↑ 10→ 2↑ 2← 2→ 2↓ 10← 1↑ 10→ 2↑ 5← 5→
2↓ 10← 1↑ 9→ 9← 2↓ 10→ 2↑ 10← 1↑ 8→ 8← 2↓ 10→ 2↑ 1← 1→ 2↓ 10← 2↑ 9→ 9← 1↑ 9→ 1↑ 1→ 4↓ 10← 2↓ 10→
10← 2↑ 10→ 2↓ 10← 2↓ 11→ 1↑ 1← 1→ 2↓ 11← 2↑ 8→ 1↑ 1→ 1↓ 2→ 2↓ 3← 1↑ 2→ 10← 2↓ 11→ 2↑ 11← 2↑ 6→
```

The code found a solution—even if it's probably not the shortest— which means I reached my goal.

# Part 7: hitting the brute-force wall

Following the sixth part, where I started dealing with rule changing, I'll add some elements about the "YOU" rule.

## "Baba is you" and nothing else

The second level start with the "Baba is you" rule, and doesn't require this rule to be changed. This means that preventing that the rule is changed is a way to implement the "YOU" that is sufficient to ensure the solution is correct.

*State.java*

```
@Nullable byte[] process() {
  processRules();
  if((youTilesMask & Tiles.BABA_MASK) == Tiles.EMPTY) {
    return null;
  }
```

Unfortunately, this is what happens:

```
2019-05-26 16:02:20.446 babaisyousolver/levels/01 Reading level
2019-05-26 16:02:20.556 babaisyousolver/levels/01 Solving level
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at net.archiloque.babaisyousolver.State.addMovement(State.java:178)
    at net.archiloque.babaisyousolver.State.tryToGo(State.java:107)
    at net.archiloque.babaisyousolver.State.process(State.java:77)
    at net.archiloque.babaisyousolver.Level.solve(Level.java:80)
    at net.archiloque.babaisyousolver.App.processLevel(App.java:50)
    at net.archiloque.babaisyousolver.App.main(App.java:29)

Process finished with exit code 1
```

## Baba is stronger than the brute-force

The problem is that the constraint added by this simple rule is enough to show the limit of the current implementation: storing all the past states fills the memory.

Adding more memory to the problem wouldn't be enough, of even storing the states into an external database: trying to explore all the possible combinations—brute-forcing the game—doesn't work here because there's too many possible moves to explore.

To be sure it's not a bug, the code can solve a smaller version of the same level:

*levels/01/content.txt*

```
F
I  f
 !
wwww
B W
I Ib
Y S
```

*levels/01/solution.txt*

```
1← 1→ 1↓ 1← 1→ 3↑ 3← 1↑ 1→ 1↓ 2→ 2↓ 1← 1↑ 1→ 1↑ 1← 1→ 1↑ 1← 1→ 2↓ 2← 1↑ 1← 1↑ 2→ 1← 2↓ 2→ 2↑ 1←
2↑ 1→ 2↓ 1← 2↑ 1← 1↓ 2→ 2↓ 3← 1↑ 1→ 1↓ 2→ 1↓ 2← 3↑ 2→ 1↓ 1← 1→ 2↑ 2← 2↓ 2→ 2↓ 2← 1↑ 1↓ 1→ 2↓ 1→
2↑ 2← 1↑ 1← 1↑ 1→ 1↓ 1→ 1↓ 1→ 1↑ 2← 2↑ 2→ 1↓ 1↑ 1← 1↓ 1→ 1↓ 1← 1→ 2↑ 2← 1↓ 1← 1↓ 1→ 1↓ 1→ 1↓ 1→
2↑ 1← 1→ 1↑ 3← 1↓ 1→ 1↓ 1→ 1↑ 1← 2↑ 1→ 2↓ 1→ 2↑ 2← 2↓ 2→ 2↓ 1← 1↑ 1→ 1↑ 1← 1↓ 1← 1↑ 1→ 1↑ 1← 2→
1↑
```

Adding the missing elements, like a complete implementation of the “YOU” rule, would add event more possibilities.

# Baba says: it’s time to start over

I started with the brute-force approach for two reasons:

1. some games with a low number of possibilities can be solved with it, for example RGB Express
2. even if it doesn’t work at the end, starting with this approach provides a way to start by focusing on modeling the game

The modeling part worked: I’ve found a way to use bit fields to manage the game states and the changing rules. I’ll be able to rely on this knowledge and parts of the code for the next step so I don’t restart from scratch.

# Part 8: learning to play

Following the seventh part, where I hit the wall of complexity, it's time to restart afresh.

The system I worked until now didn't really know how to play: it knew the rules of the game, and tried to play all possible moves one after another until it found a solution.

This approach is easy to code, but doesn't work when the game has many possible moves.

In some games, a naive approach like this can be optimized by trying to eliminate some possibilities without exploring all the possible moves. For example if it's possible to detect that a move makes the level's exit unreachable.

Unfortunately it's not the case for Baba is you, so the solution is learn the program how to play.

## Teach a machine to play

Learning the program to play means to implement a goal-oriented solver.

When a human play the game, they check the resources available on the board, then try to find an idea for a solution ("Maybe I move this rock here so I can reach the flag there?"), then try to find the mean to implement this idea.

If the idea can't be implemented, they look for another idea.

Sometimes solving the a requires sub-goals: if reaching the flag is not possible as is because a wall is blocking the path, is it possible to change the rule first so Baba can move across the wall?

The target is to create a system which rely on the same process.

Compared to the naive approach, cutting the solving into sub-goals should decrease the number of states to explore because the possible movements for each sub-goals is low enough to be manageable. For example trying to reach a specific cell without changing any rule.

This means this approach should have a better scalability for complex levels.

Its drawback is that managing all the cases requires much more code.